Algoritmos y Programación en Pascal

Cristóbal Pareja Flores
Manuel Ojeda Aciego
Ángel Andeyro Quesada
Carlos Rossi Jiménez

Algoritmos y Programación en Pascal

A nuestros compañeros y alumnos

Índice

Presentación

Este libro trata sobre métodos de resolución de problemas mediante el desarrollo de algoritmos y estructuras de datos, desde el principio y paso a paso, y su materialización en programas de computador.

Desde luego, no es el primer libro sobre este tema; de hecho, ha habido en los últimos quince años un gran aluvión de textos sobre algoritmos y sobre programación. La razón para ello ha sido sin lugar a dudas doble: por un lado, la difusión que estos temas han tenido y siguen teniendo, integrándose en los estudios más diversos; por otro, la evolución que está experimentando el desarrollo de algoritmos y programas, pasando de ser un arte (reinventado por cada programador a base de técnicas personales, estrechamente vinculadas con su lenguaje de programación) a una actividad más científica, metodológica y disciplinada.

Por consiguiente, resulta necesario aclarar cuál es el enfoque adoptado en este libro. Examinando la bibliografía existente actualmente sobre programación a un nivel introductorio permite afirmar las siguientes conclusiones:

- Una parte importante de los libros existentes han adoptado un enfoque práctico puro, no metodológico, que es el más tradicional, y aún subsiste en demasiados libros. Se confunde la enseñanza de la programación con la de un lenguaje concreto, ofreciendo muchas veces un mero "manual de referencia" del lenguaje elegido. Bajo el atractivo de los llamativos resultados inmediatos (programas que funcionan), este enfoque ignora la base conceptual y metodológica necesaria, y propicia los peores hábitos de programación, que son además difíciles de erradicar.

- Otra postura extrema se centra en el análisis y desarrollo de soluciones algorítmicas puras, de forma independiente de cualquier lenguaje de programación. Esta independencia permite ignorar las peculiaridades de los lenguajes reales, yendo a los conceptos; sin embargo, esa independencia de los lenguajes de programación es a nuestro entender innecesaria e inconveniente en los primeros pasos, ya que obliga al aprendiz de la programación a estudiar aparte los detalles concretos del lenguaje de programación con que necesariamente debe desarrollar sus prácticas.

En cambio, encontramos este enfoque interesante en niveles superiores de la enseñanza de la programación, donde interesa concentrarse en los conceptos, más difíciles y donde ya no supone obstáculo alguno expresar las ideas en cualquier lenguaje de programación.

El enfoque adoptado en este libro recoge ambos aspectos: por un lado, viene a cubrir la necesidad de un enfoque metodológico en el aprendizaje y en el ejercicio de la programación, pero también la necesidad de experimentar con programas concretos, expresarlos en un lenguaje real y hacerlos funcionar con un traductor concreto. En resumen, intentamos compaginar las ventajas de los enfoques anteriores, presentando la base conceptual y metodológica necesaria para desarrollar los algoritmos de forma razonada y disciplinada, sin olvidar por ello la conveniencia de expresarlos en un lenguaje de programación, materializándolos y experimentando con ellos, y que el lector ha de ser instruido también en esta tarea.

En relación con el enfoque metodológico que se impone actualmente, se considera necesario atender a la corrección de los programas. El tratamiento que se le da en la literatura ha llevado nuevamente a dos posturas artificialmente extremas:

- Algunos autores ignoran completamente el estudio de la corrección, contentándose con algunas comprobaciones para deducir que un programa es correcto.

- En cambio, otros adoptan un tratamiento exhaustivo, utilizando técnicas formales de especificación o verificación.

A nuestro entender, es incuestionable la importancia de garantizar que los programas desarrollados funcionarán de la forma deseada. Sin embargo, la verificación formal de los programas de cierto tamaño es impracticable. Por ello, asumimos de nuevo una posición intermedia y realista consistente en los siguientes planteamientos:

- Plantear el desarrollo de programas correctos con el empleo de técnicas semiformales.

- Limitar el estudio de la corrección a los elementos que resulten delicados, bien por su dificultad o por su novedad.

- Atender a la corrección de los algoritmos durante su desarrollo en lugar de *a posteriori*. Esta idea resulta ser una ayuda esencial en el aprendizaje de la programación.

En resumidas cuentas, este libro va dirigido a aquéllos que desean introducirse en la programación, con una base sólida, con una buena metodogía de diseño y desarrollo de programas correctos y con hábitos disciplinados desde una perspectiva realista y pragmática. Se presentan las técnicas con un cierto nivel de abstracción para identificar los conceptos esenciales e independientes del lenguaje de programación empleado, y al mismo tiempo se aterriza expresando estas técnicas en un lenguaje concreto.

El lenguaje escogido para estas implementaciones ha sido Pascal. Esta elección se debe a que este lenguaje es simple y tiene una sintaxis sencilla, que hace que sea fácil de aprender, y al mismo tiempo es lo bastante completo como para plasmar las diferentes técnicas y métodos necesarios en programas de complejidad media-alta. Esto lo hace una herramienta pedagógica idónea para el aprendizaje de la programación. A todo esto hay que sumar las numerosas implementaciones existentes y su accesibilidad, así como su evolución y continua puesta al día para permitir técnicas de programación actuales (por ejemplo, modular u orientada a los objetos) y su gran difusión y aceptación en el ámbito académico.

Organización del libro

El libro está estructurado en siete partes. En cada una de ellas se estudian las técnicas y mecanismos nuevos, conceptualmente primero, detallando luego su tratamiento en Pascal y, finalmente, compaginando ambas facetas con el aspecto metodológico. Cada tema se ha dividido en varios capítulos para evitar una excesiva fragmentación. En cada capítulo se ha incluido una lista de ejercicios propuestos de dificultad aproximadamente creciente. Al final de cada tema se desarrolla un ejemplo completo pensado para mostrar a la vez los aspectos más destacados del mismo, así como unas pocas referencias comentadas que se sugieren como lecturas complementarias o de consulta.

Contenido

El contenido se ha seleccionado partiendo de las directrices señaladas en [DCG+89] y [Tur91]. Incluye los contenidos cursos CS1 y CS2 [GT86, KSW85] salvo los aspectos de organización de computadores, que se estudian en [PAO94], de los mismos autores que este libro.

En el primer tema se presentan, entre otros, los conceptos esenciales de algoritmo, dato y programa. Se introduce el lenguaje Pascal y la estructura de los programas escritos en él, así como los elementos básicos del lenguaje. Se incluyen

algunos programas sencillos, y se adelantan la técnica descendente de diseño de programas y algunos apuntes sobre la corrección.

El segundo tema se dedica a la programación estructurada. Se pone especial énfasis en el diseño descendente o por refinamientos sucesivos partiendo de especificaciones escritas en pseudocódigo, y se muestra cómo compaginar esta técnica con la derivación de programas correctos.

En el tercer tema se estudian los subprogramas. Al igual que en el tema anterior, se detalla cómo enfocar la corrección en el uso de esta técnica. Se concluye con un capítulo de introducción a la recursión.

En la mayoría de los programas no basta con los tipos de datos básicos, sino que es necesario que el programador defina otros más complejos. A ello se dedica el cuarto tema.

El quinto tema estudia las técnicas propias de la gestión de memoria dinámica. Se justifica su necesidad, y se presenta su principal aplicación, que es la definición de estructuras de datos recursivas.

El sexto tema introduce tres aspectos avanzados: la programación con tipos abstractos de datos, el coste de los algoritmos y los principales esquemas de diseño de algoritmos. Aunque, ciertamente, su estudio en profundidad rebasa un primer curso, es frecuente introducir –o siquiera mencionar– sus ideas básicas. Por supuesto, siempre es altamente recomendable consultar otras referencias (nosotros mismos las seleccionamos para cada tema), pero también es cierto que el alumno se ve obligado con frecuencia a usar varios textos básicos para cubrir diferentes partes de la materia. Justamente, estos últimos capítulos se incluyen para que el lector interesado se pueda asomar a ellos sin verse obligado a consultar los capítulos introductorios de otros libros.

Finalmente se incluyen tres apéndices: en el primero se introducen un par de aspectos complementarios para un primer curso de programación (el paso de subprogramas como parámetros y el uso de variables aleatorias); el segundo es un prontuario de uso del entorno integrado de desarrollo Turbo Pascal; y el tercero indica algunos detalles de Turbo Pascal en que se separa del estándar, pero que son de uso frecuente.

Notación empleada

En la lectura de este texto se encontrarán fragmentos escritos en distintos lenguajes:

- En castellano puro, donde se ha usado este tipo de letra.
- En Pascal, para lo que se ha elegido el `teletipo`, salvo las palabras reservadas, que van en **negrita**.

También se ha empleado el `teletipo` para indicar las salidas y entradas de datos, porque es el tipo de letra más parecido al que aparece en el monitor.

- En seudocódigo, que se expresa con *letra cursiva*.
- En lenguaje matemático, en que se usan los símbolos usuales.
- Otros símbolos especiales:

$\sqcup$	El espacio en blanco (véase la página 206)
$\rightsquigarrow$	Uno o más pasos al evaluar una expresión (véase la página 30)
$\bullet$	Fin de archivo (véase la página 57)
$\hookleftarrow$	Fin de línea (véase la página 57)

Advertencia para el alumno

No existe un método general para resolver problemas mediante algoritmos; por ello, es de gran importancia estudiar las técnicas de forma gradual, yendo de lo fácil a lo difícil y vinculándolas con situaciones ampliamente conocidas.

Nosotros pretendemos haber cumplido con este objetivo, proporcionando ejemplos de fácil comprensión y ejercicios a la medida de lo explicado. Y eso es lo bueno. Lo malo es que el alumno puede percibir la sensación de comprenderlo todo a la velocidad que lee... y aquí reside el peligro. Porque una mera lectura del texto, o incluso una lectura atenta, no basta para asimilarlo, adquiriendo las técnicas necesarias para resolver otros problemas de dificultad similar a la de los planteados. Es preciso adoptar una actitud crítica durante la lectura, trabajar minuciosamente con los problemas propuestos e incluso tatar de imaginar soluciones alternativas a los ejemplos dados. Y cuanto antes se acepte esa realidad, mejor que mejor.

Agradecimientos

Muchas personas han contribuido de diferentes maneras a que este libro sea lo que es. En primer lugar, debemos a nuestros alumnos de estos años su ayuda, aun sin saberlo, porque ellos han sido la razón de que emprendiéramos este trabajo, y porque sus preguntas y comentarios, día a día, tienen una respuesta en las páginas que siguen. En segundo lugar, debemos a muchos de nuestros compañeros su aliento y apoyo, tan necesario cuando uno se enfrenta a un trabajo de esta envergadura. Y por ello, lo dedicamos a ambos, alumnos y compañeros.

De un modo muy especial, deseamos expresar nuestro agradecimiento a Juan Falgueras Cano, Luis Antonio Galán Corroto y Yolanda Ortega y Mallén por su cuidadosa lectura de la primera versión completa del manuscrito, y por sus

valiosos comentarios y sugerencias. No podemos olvidar tampoco la ayuda de Manuel Enciso García-Oliveros en los últimos retoques, así como su proximidad y apoyo durante todo el tiempo que nos ha ocupado este trabajo.

Por último, deseamos expresar nuestra gratitud y cariño a José Luis Galán García, que ha dejado en este libro muchas horas.

Tema I

Algoritmos e introducción a Pascal

Capítulo 1

Problemas, algoritmos y programas

En este primer capítulo se trata la resolución de problemas por medio de un computador. Puesto que éste no es más que un mero ejecutor de tareas, es fundamental conocer un método de resolución del problema en cuestión, esto es, un algoritmo. La escritura de este algoritmo como un conjunto de órdenes comprensibles por el computador es lo que se llama programa.

1.1 Solución de problemas mediante programas

Los computadores desempeñan una gran variedad de tareas, liberando así al hombre de tener que realizarlas personalmente. Para ello, es preciso enseñar al computador cuál es su trabajo y cómo llevarlo a cabo, esto es, programarlo,

dándole instrucciones precisas (programas) en un lenguaje que comprenda (Pascal, Modula-2, C, etc.). Una vez "aprendido" el programa, el computador seguirá ciegamente sus instrucciones cuantas veces sea requerido.

Precisamente, la tarea de la programación consiste en describir lo que debe hacer el computador para resolver un problema concreto en un lenguaje de programación. Sin embargo, el programa es solamente el resultado de una serie de etapas que no se pueden pasar por alto. Hablando en términos muy amplios, se identifican de momento las siguientes fases:

1. Análisis del problema, estableciendo con precisión lo que se plantea.
2. Solución conceptual del problema, describiendo un método (algoritmo) que lo resuelva.
3. Escritura del algoritmo en un lenguaje de programación.

En la primera fase es corriente partir de un problema definido vagamente, y el análisis del mismo consiste en precisar el enunciado, identificando los datos de partida y los resultados que se desean obtener. La descripción precisa de un problema se llama *especificación*. Con frecuencia, el lenguaje natural no basta para lograr la precisión deseada, por lo que se recurre en mayor o menor medida a lenguajes formales, como la lógica o las matemáticas. Supongamos por ejemplo que se plantea el problema de dividir dos números. En primer lugar, se necesita saber si se trata de la división entera o de aproximar el resultado con decimales y, en ese caso, hasta dónde. Pongamos por caso que interesa la división euclídea. Una descripción precisa deberá tener en cuenta que los datos son dos enteros (llamémosles *dividendo* y *divisor*, como es usual), de los que el segundo es no nulo. El resultado es también un par de enteros (llamémosles *cociente* y *resto*, como siempre) tales que

$$dividendo = divisor * cociente + resto$$

Pero eso no es todo: si nos contentamos con ese enunciado, para todo par de enteros $(dividendo, divisor)$, el par $(0, dividendo)$ siempre es una solución. Por eso, hay que añadir que *resto* debe ser además tal que $0 \leq resto < divisor$.

Con este pequeño ejemplo se pretende resaltar la importancia de analizar bien el problema planteado, definiendo con precisión los requisitos que deben verificar los datos y las condiciones en que deben estar los resultados.

Sin embargo, en esta fase sólo se ha estudiado *qué* se desea obtener, y no *cómo* lograrlo. Éste es el cometido de la segunda etapa: describir un método (*algoritmo*) tal que partiendo de datos apropiados lleve sistemáticamente a los resultados descritos en la especificación. Del concepto de algoritmo nos ocupamos

1 Principio → 2	2 ¿Está abierto el plazo de matrícula? sí → 4 no → 3	3 Esperar a mañana → 2
4 Comprar impresos de matriculación → 5	5 Leer instrucciones. ¿Tengo alguna duda? sí → 6 no → 7	6 Preguntar dudas en Secretaría → 7
7 Rellenar el sobre y pagar en el banco → 8	8 Entregar el sobre → 9	9 Fin

Figura 1.1.

en el siguiente apartado. Es evidente que sólo se puede confiar en un algoritmo si ha superado con éxito determinado control de calidad: la primera e inexcusable exigencia es que sea correcto; esto es, que resuelva el problema especificado. Cuando un problema admita varios algoritmos como solución, convendrá disponer de criterios para escoger; y cuando un problema no tenga solución, no resulta sensato buscar un algoritmo para resolverlo. Estos aspectos los estudiamos en el apartado 1.3.

Finalmente, para que un computador resuelva problemas hay que escribir el algoritmo en un lenguaje de programación; de ello hablaremos en el apartado 1.4.

1.2 Concepto de algoritmo

Los conceptos de algoritmo y de método son parecidos: los métodos para efectuar procesos forman parte de las costumbres o rutinas que el hombre aprende un día y luego repite de manera inconsciente, sin reparar ya en las acciones, más sencillas, que integran el proceso (por ejemplo andar, leer o conducir). Por eso el concepto de algoritmo suele compararse a otros como método o rutina de acciones.

La secuencia de pasos de la figura 1.1 describe un método para efectuar la matrícula en la universidad.

Sin embargo, el término "algoritmo" tiene connotaciones más formales que cualquier otro debido a su origen: se trata de una acomodación al castellano del

nombre de Muḥammad ibn Mūsā al-Jwārīzmī, matemático persa que popularizó su descripción de las cuatro reglas (algoritmos) de sumar, restar, multiplicar y dividir.

1.2.1 Una definición de algoritmo

Hablando informalmente, un *algoritmo* es la descripción precisa de los pasos que nos llevan a la solución de un problema planteado. Estos pasos son, en general, acciones u operaciones que se efectúan sobre ciertos objetos. La descripción de un algoritmo afecta a tres partes: entrada (datos), proceso (instrucciones) y salida (resultados).[1] En este sentido, un algoritmo se puede comparar a una función matemática:

$$\underset{(algoritmo)}{+} \quad : \quad \underset{(entrada)}{\mathbb{Z} \times \mathbb{Z}} \quad \underset{(proceso)}{\longrightarrow} \quad \underset{(salida)}{\mathbb{Z}}$$

Incluso en algoritmos no matemáticos es fácil identificar las tres partes: entrada, proceso y salida. Así ocurre, por ejemplo, en las instrucciones para hacer la declaración de la renta.

Características de los algoritmos

La descripción de algoritmo que se ha dado es algo imprecisa. Una caracterización más completa debería incluir además los siguientes requisitos:

1. *Precisión*

   Un algoritmo debe expresarse de forma no ambigua. La precisión afecta por igual a dos aspectos:

   (a) Al *orden* (encadenamiento o concatenación) de los pasos que han de llevarse a cabo.

   (b) Al *contenido* de las mismas, pues cada paso debe "saberse realizar" con toda precisión, de forma automática.

   Por lo tanto, una receta de cocina puede ser considerada como un método, pero carece de la precisión que requieren los algoritmos debido al uso de expresiones como *añadir una pizca de sal*, porque ¿qué debe entenderse por una pizca?

[1] Es habitual llamar "datos" a la entrada y "resultados" a la salida, aunque el concepto de dato es más amplio, y abarca a toda la información que maneja un algoritmo, ya sea inicialmente o a su término, así como también durante el transcurso de su utilización.

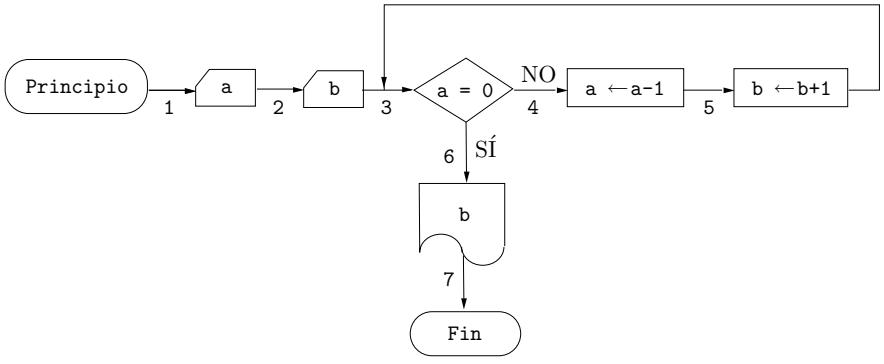


Figura 1.2. Diagrama de flujo de la "suma lenta".

2. *Determinismo*

   Todo algoritmo debe responder del mismo modo ante las mismas condiciones.

   Por lo tanto, la acción de barajar un mazo de cartas *no* es un algoritmo, ya que es y debe ser un proceso no determinista.

3. *Finitud*

   La descripción de un algoritmo debe ser finita.

Un primer ejemplo

Consideremos el ejemplo que, expresado gráficamente,[2] aparece en la figura 1.2. El algoritmo descrito tiene por objetivo sumar dos cantidades enteras. Si se anotan esas cantidades inicialmente en sendas casillas (a las que llamaremos a y b para abreviar), este método consiste en ir pasando de a a b una unidad cada vez, de forma que, cuando $a = 0$, el resultado será el valor de b.

Vemos un ejemplo de su funcionamiento con los datos 2 y 3 en la figura 1.3. (Los números se han incluido para seguir mejor la evolución de los cálculos.)

Se observa que la descripción del algoritmo que se ha dado es, en efecto, precisa (cada paso está exento de ambigüedad, así como el orden en que se debe efectuar) y determinista (el efecto de cada paso es siempre el mismo para unos datos concretos cualesquiera). Estas dos características son una consecuencia del lenguaje escogido para expresar el algoritmo.

[2]El lenguaje empleado es el de los diagramas de flujo, que estudiaremos en el capítulo 7, apartado 7.2.1. Esperamos que, debido a su sencillez, el funcionamiento de este ejemplo resulte comprensible directamente.

Posición	Datos pendientes	Resultados emitidos	Var a	Var b
1	[2, 3]	[ ]	?	?
2	[3]	[ ]	2	?
3	[ ]	[ ]	2	3
4	[ ]	[ ]	2	3
5	[ ]	[ ]	1	3
3	[ ]	[ ]	1	4
4	[ ]	[ ]	1	4
5	[ ]	[ ]	0	4
3	[ ]	[ ]	0	5
6	[ ]	[ ]	0	5
7	[ ]	[5]	0	5

Figura 1.3. Ejecución de la suma lenta con los datos 2 y 3.

En cambio, si bien es cierto que, con los datos del ejemplo, se ha obtenido una solución, si se examina con detenimiento se verá que ese "algoritmo" no siempre termina para dos cantidades enteras cualesquiera (v. g. -3 y -1). De hecho, la terminación es una característica escurridiza de la que hablaremos más tarde (véase el apartado 1.3.1).

1.2.2 Una definición formal de algoritmo

La caracterización hecha hasta ahora de los algoritmos es satisfactoria a efectos prácticos. Más concreta aún resulta a la vista de un lenguaje algorítmico como el de los diagramas de flujo, que deja entrever dos aspectos:

- El mantenimiento de unas variables (a y b) y de unas posiciones $(1, \ldots, 7)$, a lo que llamamos *estado*. Por ejemplo, cada fila de la tabla de la figura 1.3 representa un estado en la evolución del algoritmo 1.2.
- La descripción de *transiciones* entre estados, que vienen dadas por el propio diagrama.

Estos aspectos de los algoritmos están ausentes de la primera definición, por lo que puede resultar aún algo incompleta; surge pues la necesidad de una definición más formal:

Definición: Un algoritmo es una cuádrupla que comprende los siguientes elementos:

1. El conjunto de los *estados* (llamémosle E) que pueden presentarse en todo momento durante el cálculo.

Un estado viene dado por una tupla, incluyendo:

- los valores de las variables que entran en juego;
- los datos sin leer y los resultados emitidos, y
- la marca, identificando la posición del algoritmo en la que se da este estado de los cálculos.

Es decir, un estado se puede expresar así:

<Datos por leer; Resultados emitidos; Variables; Posición>

2. La identificación de los *estados iniciales*, $I \subset E$, o estados posibles al comienzo del algoritmo.

3. La identificación de los *estados finales*, $F \subset E$, como posibles estados al terminar el algoritmo.

4. Una *función de transición* entre estados,

$$t : E \longrightarrow E$$

que describe el efecto de cada paso del cómputo asociado al algoritmo. Esta función deberá:

- Estar definida dentro de E, esto es, para cualquier $e \in E$ debemos tener que $t(e) \in E$. Así, las transiciones entre estados son precisas y deterministas.
- A partir de cualquier estado inicial, la función de transición t debe llevar a un estado final en un número finito de pasos, formalmente: para cualquier $e \in I$ existe un $k \in \mathbb{N}$ tal que

$$\overbrace{t(t(\cdots t(e)\cdots))}^{t \text{ se aplica } k \text{ veces}} \in F$$

y, además, no tiene efecto alguno sobre los estados finales, es decir: para cualquier $e \in F$ ocurre que $t(e) = e$. De aquí se obtiene la característica de finitud.

Siguiendo con el algoritmo del diagrama de flujo de la figura 1.2, identificamos los siguientes estados:

$$\begin{array}{rlllllllll}
E = \{ & < & 1 & ; & [d_1, d_2] & ; & [\,] & ; & — & > \\
& < & 2 & ; & [d_2] & ; & [\,] & ; & a \equiv a_0 & > \\
& < & p & ; & [\,] & ; & [\,] & ; & a \equiv a_0, b \equiv b_0 & > \\
& < & 7 & ; & [\,] & ; & [r] & ; & — & > \; \}
\end{array}$$

donde $d_1, d_2, a, b, r \in \mathbb{N}, p \in \{3, \ldots, 6\}$, y siendo

$$I = \{< 1; [d_1, d_2]; [\,]; — >\}$$

y

$$F = \{< 7; [\,]; [r]; — >\}.$$

La función de transición t es la siguiente:

$$\begin{array}{lcll}
t(< 1; [d_1, d_2]; [\,]; — >) & = & < 2; [d_2]; [\,]; a \equiv d_1 > & \\
t(< 2; [d_2]; [\,]; a \equiv d_1 >) & = & < 3; [\,]; [\,]; a \equiv d_1, b \equiv d_2 > & \\
t(< 3; [\,]; [\,]; a \equiv a_0, b \equiv b_0 >) & = & \begin{cases} < 6; [\,]; [\,]; b \equiv b_0 > & \text{si } a = 0 \\ < 4; [\,]; [\,]; a \equiv a_0, b \equiv b_0 > & \text{si } a \neq 0 \end{cases} & \\
t(< 4; [\,]; [\,]; a \equiv a_0, b \equiv b_0 >) & = & < 5; [\,]; [\,]; a \equiv a_0 - 1, b \equiv b_0 > & \\
t(< 5; [\,]; [\,]; a \equiv a_0, b \equiv b_0 >) & = & < 3; [\,]; [\,]; a \equiv a_0, b \equiv b_0 + 1 > & \\
t(< 6; [\,]; [\,]; b \equiv b_0 >) & = & < 7; [\,]; [b_0]; — > & \\
t(< 7; [\,]; [r]; — >) & = & < 7; [\,]; [r]; — > &
\end{array}$$

Desde el punto de vista del usuario de un algoritmo, se puede considerar un algoritmo como una caja opaca cuyos detalles internos se ignoran, aflorando sólo la lectura de los datos y la escritura de los resultados. Estos aspectos observables desde el exterior se llaman frecuentemente la *interfaz externa*. No obstante, el mecanismo interno interesa al autor de algoritmos, esto es, al programador. Para atender esta necesidad, algunos entornos de programación permiten "trazar" programas, con lo que el programador puede ver evolucionar el estado interno durante la marcha de su programa con el grado de detalle que desee (véanse los apartados 5.4.1 y C.2.6). Esta posibilidad es interesante para depurar los programas, esto es, para buscar los posibles errores y subsanarlos.

Para terminar este apartado, debemos decir que el esquema de funcionamiento descrito a base de transiciones entre estados se conoce con el nombre de *modelo de von Neumann* (véase el apartado 7.2.1 de [PAO94]). Se dice que este modelo es *secuencial*, en el sentido de que los pasos se efectúan uno tras otro. De hecho, no es posible acelerar el proceso efectuando más de un paso a un tiempo, porque cada paso tiene lugar desde el estado dejado por el paso anterior. Este "cuello de botella" es un defecto propio de las máquinas de von Neumann.

Cualidades deseables de un algoritmo

Es muy importante que un algoritmo sea suficientemente general y que se ejecute eficientemente. Veamos con más detalle qué se entiende por *general* y por *eficiente*:

1. *Generalidad*

   Es deseable que un algoritmo sirva para una clase de problemas lo más amplia posible. Por ejemplo, la clase de problemas "resolver una ecuación de segundo grado, $a + bx + cx^2 = 0$" es más general que la consistente en "resolver ecuaciones de primer grado, $a + bx = 0$".

2. *Eficiencia*

   Hablando en términos muy generales, se considera que un algoritmo es tanto más *eficiente* cuantos menos pasos emplea en llevar a cabo su cometido. Por ejemplo, para hallar la suma de dos números naturales, la regla tradicional que se aprende en enseñanza primaria es más eficiente que el rudimentario procedimiento de contar con los dedos, de uno en uno. Este tema se revisará en el apartado 1.3.3, y será tratado con mayor detalle en el capítulo 18.

Estas dos cualidades no son siempre conciliables, por lo que frecuentemente hay que optar por una solución en equilibrio entre ambas cuando se diseña un algoritmo. Por ejemplo, un algoritmo que estudia y resuelve "sistemas de ecuaciones" es más general que uno que resuelve "sistemas de ecuaciones lineales", y éste a su vez es más general que otro que sólo considera "sistemas de dos ecuaciones lineales con dos incógnitas". Sin embargo, a mayor generalidad se tiene también una mayor complejidad puesto que hay que tratar más casos y no se pueden aplicar algoritmos específicos. Por consiguiente, si un buen número de situaciones puede resolverse con un algoritmo rápido aunque poco general, es preferible adoptar éste.

1.3 Aspectos de interés sobre los algoritmos

1.3.1 Computabilidad

Con el aumento de potencia y el abaratamiento de los computadores, cada vez se plantean aplicaciones más sorprendentes y de mayor envergadura, capaces de resolver problemas más generales. Da la impresión de que cualquier problema que se plantee ha de ser *computable* (esto es, ha de tener una solución algorítmica), sin otra limitación que la potencia de la máquina ejecutora.

Sin embargo, esto no es así. Por rotunda que pueda parecer esta afirmación, se debe aceptar que

hay problemas no computables

La cuestión que se plantea es algo delicada: adviértase que, si no se conoce algoritmo que resuelva un problema planteado, sólo se puede asegurar "que no

se conoce algoritmo que resuelva ese problema"; en cambio, establecer que un problema "no es computable" requiere demostrar que *nunca* se podrá encontrar ningún algoritmo para resolver el problema planteado.

Los siguientes son ejemplos de problemas no computables:

- *Décimo problema de Hilbert.*

  Resolver una ecuación diofántica con más de una incógnita.

  Esto significa encontrar soluciones enteras de una ecuación de la forma $P(x_1, x_2, \ldots) = 0$, donde P es un polinomio con coeficientes enteros.

- *Problema de la parada.*

  Determinar si un algoritmo a finaliza o no cuando opera sobre una entrada de datos d:

  $$\textsc{Stop}(a, d) = \begin{cases} \text{Sí} & \text{si } a(d) \downarrow \\ \\ \text{No} & \text{si } a(d) \uparrow \end{cases}$$

  donde $a(d) \downarrow$ (resp. $a(d) \uparrow$) expresa que el algoritmo a, aplicado al dato d, sí para (resp. no para).

Examinaremos con más atención el problema de la parada, y después veremos cuán escurridizo puede resultar determinar si un algoritmo particular para o no considerando un sencillo ejemplo. Claro está que esto no demuestra nada salvo, en todo caso, nuestra incapacidad para analizar este algoritmo en concreto. Por ello, incluimos seguidamente una demostración de que es imposible hallar un algoritmo capaz de distinguir entre los algoritmos que paran y los que no.

Obsérvese que se plantea aquí estudiar "algoritmos que operan sobre algoritmos, vistos como datos". Por extraño que parezca al principio, en Computación se desarrollan frecuentemente programas que manejan otros programas con diversos fines.

El problema de la parada no es computable

El problema de la parada, definido más arriba, se puede expresar como sigue: ¿se puede examinar cualquier algoritmo y decidir si para? Demostraremos que no existe, ni puede existir, el algoritmo STOP que distinga si un algoritmo a para cuando se aplica a los datos d.

Procederemos por reducción al absurdo: Suponiendo que existe ese algoritmo (descrito más arriba como STOP), a partir de él es posible construir otro similar, STOP', que averigüe si un algoritmo a para cuando se aplica "a su propio texto":

$$\textsc{Stop'}(p) = \textsc{Stop}(p, p) = \begin{cases} \text{Sí} & \text{si } p(p) \downarrow \\ \text{No} & \text{si } p(p) \uparrow \end{cases}$$

Y entonces, también se puede definir el siguiente algoritmo a partir del anterior:

$$\textsc{Raro}(p) = \left\{ \begin{array}{ll} \uparrow & \text{si } \textsc{Stop'}(p) = \text{Sí} \\ \text{No} & \text{si } \textsc{Stop'}(p) = \text{No} \end{array} \right\} = \left\{ \begin{array}{ll} \uparrow & \text{si } p(p) \downarrow \\ \text{No} & \text{si } p(p) \uparrow \end{array} \right.$$

Veamos que el algoritmo Raro tiene un comportamiento verdaderamente extraño cuando se aplica a sí mismo:

$$\begin{aligned} \textsc{Raro}(\textsc{Raro}) &= \left\{ \begin{array}{ll} \uparrow & \text{si } \textsc{Stop'}(\textsc{Raro}) = \text{Sí} \\ \text{No} & \text{si } \textsc{Stop'}(\textsc{Raro}) = \text{No} \end{array} \right. \\ &= \left\{ \begin{array}{ll} \uparrow & \text{si } \textsc{Raro}(\textsc{Raro}) \downarrow \\ \text{No} & \text{si } \textsc{Raro}(\textsc{Raro}) \uparrow \end{array} \right. \end{aligned}$$

lo que resulta obviamente imposible.

La contradicción a que hemos llegado nos lleva a rechazar la hipótesis inicial (la existencia de un algoritmo para el problema de la parada), como queríamos demostrar.

Números pedrisco

Como ejemplo de la dificultad de examinar un algoritmo y decidir si concluirá tarde o temprano, consideremos la siguiente función t: $\mathbb{N} \to \mathbb{N}$:

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

Partiendo de un número natural cualquiera, le aplicamos t cuantas veces sea necesario, hasta llegar a 1. Por ejemplo,

$$3 \xrightarrow{t} 10 \xrightarrow{t} 5 \xrightarrow{t} 16 \xrightarrow{t} 8 \xrightarrow{t} 4 \xrightarrow{t} 2 \xrightarrow{t} 1 \cdots$$

Esta sencilla sucesión genera números que saltan un número de veces impredecible,

$$27 \xrightarrow{t^{111}} 1$$

de manera que cabe preguntarse si todo natural n alcanza el 1 tras una cantidad finita de aplicaciones de t, o por el contrario existe alguno que genera términos que saltan indefinidamente.[3]

Se ha comprobado, por medio de computadores, que la sucesión llega a 1 si se comienza con cualquier número natural menor que $2^{40} \simeq 10^{12}$, pero aún no se ha podido demostrar para todo n.

[3]Este problema también se conoce como *problema 3n+1*.

1.3.2 Corrección de algoritmos

El aspecto de la corrección es de crucial importancia para quien desarrolla un algoritmo. Sin embargo, es imposible detectar los posibles errores de una forma sistemática. Con frecuencia, la búsqueda de errores (*depuración*) consiste en la *comprobación* de un algoritmo para unos pocos juegos de datos. Sin embargo, eso no garantiza nada más que el buen funcionamiento del algoritmo para esos juegos de datos y no para todos los posibles. Volviendo al algoritmo de la suma lenta, la comprobación con los juegos de datos [2, 3] y [3, -1] ofrecería resultados correctos, y sin embargo el algoritmo únicamente termina cuando el primer sumando es (un entero) positivo.

Frente a la comprobación, la *verificación* consiste en la demostración del buen funcionamiento de un algoritmo con respecto a una especificación. Esto es, la verificación trata de garantizar que, para todos los datos considerados (descritos en la especificación), el algoritmo lleva a los resultados (también descritos en la especificación) deseados. Por eso, aunque frecuentemente se habla de corrección de un algoritmo, sin más, en realidad hay que decir *corrección de un algoritmo con respecto a una especificación*.

Por lo general, la verificación se basa en establecer *aserciones* (propiedades) del estado de la máquina en cada paso del algoritmo. Se profundizará en esta idea a lo largo de todo el texto. Por ejemplo, para verificar el funcionamiento del algoritmo de "suma lenta", consideremos que se hace funcionar sobre dos enteros genéricos m y n. Claramente, al llegar a la posición 3 por vez primera, $a = m$ y $b = n$. Por otra parte, en la posición 3 se tiene invariablemente la propiedad

$$a + b = m + n$$

independientemente de las vueltas que se den y de las modificaciones que se efectúen sobre a y b,[4] ya que cada unidad que se reste a a se le suma a b. Ahora bien, cuando se pasa de la posición 3 a la 6 es por ser $a = 0$, con lo que se tiene, simultáneamente, el invariante $a + b = m + n$ y $a = 0$, es decir:

$$b = m + n$$

lo que asegura que la salida es correcta... cuando ésta se produzca. Un algoritmo que ofrece una solución correcta cuando para, pero del que no sepamos si para o no, se dice *parcialmente correcto*. Ese es el caso de la suma lenta de números enteros.

Además, si inicialmente a es un número natural, es seguro que el algoritmo para, ya que la operación $a \leftarrow a - 1$ lo llevará a cero, precisamente en a vueltas del bucle. Si se asegura que un algoritmo es parcialmente correcto y que para

[4]Por ello, esta propiedad se conoce como *invariante*.

en un tiempo finito, se dice que el algoritmo es (*totalmente*) *correcto*. Ése es el caso de la suma lenta de pares de números, siendo el primero de ellos entero y positivo.

La verificación de algoritmos no es una tarea fácil. Al contrario, verificar completamente un algoritmo involucra el uso de lógica matemática, y con frecuencia resulta incluso más complicado que desarrollarlo. De ahí el interés que tiene esmerarse, desde el principio, en escribir algoritmos correctos, adquiriendo un buen estilo y esforzándose en emplear metodologías apropiadas para ello.

1.3.3 Complejidad de algoritmos

Resulta de gran interés poder estimar los recursos que un algoritmo necesita para resolver un problema. En máquinas secuenciales, estos recursos son el *tiempo* y la *memoria*.[5]

Muchas veces, un algoritmo tarda tanto en ofrecer el resultado que resulta, en realidad, inútil. Un ejemplo de esta situación se da en sistemas de control de procesos, en donde la respuesta a determinadas circunstancias debe disparar mecanismos de seguridad en un tiempo crítico (por ejemplo, en centrales nucleares). Análogamente para el espacio, es posible que la máquina en que ha de funcionar un programa disponga de una capacidad de memoria limitada y nos veamos obligados a elegir algoritmos que usen poca memoria. Por lo tanto, existen situaciones en las que si disponemos de varios algoritmos para un mismo problema, deberemos decidir cuál es el más rápido o el que menos cantidad de memoria requiere. En el capítulo 18 se ahondará en esta idea.

Como el tiempo requerido por los programas depende en gran medida de la potencia de la máquina ejecutora, es frecuente medir en "pasos" (de coste fijo) el coste del algoritmo correspondiente. Para concretar un poco, y siguiendo con el ejemplo de la suma lenta, consideremos que cada bloque o caja del diagrama es un paso y, por tanto, cuesta una unidad. Entonces, es fácil deducir que sumar los naturales m y n lleva $3m + 4$ pasos.

En efecto, llamando t_m al coste de sumar m y n, se tiene que

$$\begin{aligned} t_0 &= 4 \text{ pasos} \\ t_m &= t_{m-1} + 3, \text{ si } n \geq 1 \end{aligned}$$

pues para sumar 0 y n hay que realizar cuatro pasos (véase la figura 1.2) y si $m \neq 0$ hay que realizar tres pasos para reducir m en una unidad; resumiendo

$$t_0 = 4$$

[5]En los últimos años, también está interesando medir el número de procesadores en máquinas de procesamiento en paralelo (véase el apartado 3.5 de [PAO94]).

$$\begin{aligned} t_1 &= t_0 + 3 = 4 + 3 \cdot 1 \\ t_2 &= t_1 + 3 = 4 + 3 \cdot 2 \\ \vdots \quad &\vdots \quad \vdots \\ t_m &= t_{m-1} + 3 = 4 + 3 \cdot m \end{aligned}$$

independientemente de n.

Por otra parte, es frecuente concentrar el interés en el coste para datos grandes estudiando el comportamiento "asintótico". De este modo, es posible redondear el tiempo despreciando términos dominados por otros:

$$3m + 4 \overset{m \to \infty}{\approx} 3m$$

También es frecuente despreciar factores de proporcionalidad, absorbidos por las velocidades de proceso de máquinas con distinta potencia, con lo cual

$$3m \approx m$$

de manera que, en resumen, se dirá que el algoritmo estudiado tiene coste lineal. Naturalmente, podemos encontrarnos algoritmos con coste constante, logarítmico, lineal, cuadrático, exponencial, etc.

Hallar el coste de un algoritmo no es en general una tarea fácil. Con frecuencia requiere una fuerte base matemática. Sin embargo, también es útil comprender, desde el principio, que la potencia de los computadores no es ilimitada, y que la ejecución de los programas consume sus recursos. Ése es el interés de que merezca la pena esforzarse por estudiar y encontrar algoritmos eficientes, aunque ello requiere muchas veces sacrificar su simplicidad.

1.4 Lenguajes algorítmicos y de programación

La descripción de un algoritmo incluye organizar los datos que intervienen en el mismo, así como las acciones que se deben llevar a cabo sobre ellos. Una vez ideado el algoritmo, el modo más natural e inmediato (y también el menos formal) de expresar esa organización es redactándolo con palabras y frases del lenguaje cotidiano. En el extremo opuesto se sitúan, por su rigidez, los lenguajes de programación.

Entre la libertad, flexibilidad y ambigüedad de los lenguajes naturales y la precisión, rigidez y limitaciones de expresividad de los lenguajes de programación se sitúan los lenguajes algorítmicos. Éstos tienen las siguientes cualidades:

1. Tienden un puente entre la forma humana de resolver problemas y su resolución mediante programas de computador.

2. Tienen cierta independencia de los lenguajes de programación particulares, de modo que están libres de sus limitaciones y así los algoritmos escritos en ellos se pueden traducir indistintamente a un lenguaje de programación u otro.

Por ello, los lenguajes algorítmicos constituyen una herramienta expresiva con una libertad y flexibilidad próxima a la de los naturales y con el rigor de los de programación.

En realidad, las únicas restricciones que deberían imponerse a estos lenguajes proceden de las características que tienen los algoritmos: expresar sus acciones (*qué* deben realizar y *cuándo*) con la precisión necesaria, y que estas acciones sean deterministas.

Por consiguiente, todo lenguaje algorítmico debe poseer mecanismos con que expresar las acciones así como el orden en que han de llevarse a cabo. Las acciones, se expresan mediante instrucciones (también llamadas órdenes o sentencias), que son comparables a verbos en infinitivo: asignar. . . , leer. . . , escribir. . . y otras. La concatenación de las instrucciones expresa en qué orden deben sucederse las acciones; esto es, cómo se ensamblan unas tras otras. Los modos más usados para ensamblar órdenes son la secuencia, la selección y la repetición, y los estudiaremos en el capítulo 7.

A la vista de las observaciones anteriores es lógico cuestionar que los lenguajes naturales sean algorítmicos (debido sobre todo a la ambigüedad que es inherente a ellos). Además, resulta más práctico utilizar medios de expresión normalizados, facilitándose la comunicación entre diseñadores y usuarios de los algoritmos y las personas que los desarrollan.

Los *diagramas de flujo* constituyen uno de los lenguajes algorítmicos que mayor difusión han alcanzado, aunque su uso está menguando drásticamente en los últimos años en favor de otros métodos más apropiados desde el punto de vista formativo.[6] Entre ellos, debe citarse el *seudocódigo*, que aportará valiosas ideas para la adquisición de un buen estilo de programación y, en definitiva, para aprender cómo enfocar la resolución de problemas complicados. Seguidamente, describimos el algoritmo de la suma lenta mediante seudocódigo:

Sean $a,\ b \in \mathbb{Z}$
Leer a *y* b
Mientras $a \neq 0$, *hacer* $\begin{cases} a \leftarrow a - 1 \\ b \leftarrow b + 1 \end{cases}$
Escribir b

[6]De hecho, nosotros desaconsejamos su uso como lenguaje de desarrollo de algoritmos, limitando su estudio a mostrar, precisamente, lo que *no* es la programación estructurada (véase el apartado 7.2.1).

En la práctica, el seudocódigo se usa como un medio expresivo de camino hacia un lenguaje de programación concreto. Por tanto, es lógico que los algoritmos seudocodificados tengan un parecido notorio con los programas escritos en ese lenguaje. En concreto, compárese por ejemplo el seudocódigo anterior con el programa siguiente, escrito en Pascal:

```
Program SumaLenta (input, output);
  {Se suman dos enteros, pasando unidades de uno a otro}
  {PreC.:  input = [m n], enteros}
  var
    a, b: integer;
begin
  ReadLn (a, b);
    {Inv.:  a + b = m + n}
  while a <> 0 do begin
    a:= a-1;
    b:= b+1
  end; {while}
  WriteLn(b)
end.  {SumaLenta}
```

1.5 Desarrollo sistemático de programas

En los últimos años, las aplicaciones comerciales se han hecho cada vez más grandes y complejas y, por tanto, su desarrollo resulta cada vez más caro, lento y sujeto a numerosos errores. Este problema fue tremendamente importante hacia los años sesenta (se hablaba entonces de "crisis del *software*"), y para tratar de solucionarlo surgió entonces la *Ingeniería del software*, que considera el desarrollo de programas y aplicaciones como un proceso productivo donde se aplican técnicas de ingeniería. El desarrollo del *software* se organiza en fases, que en conjunto se conocen como el ciclo de vida.[7] Son las siguientes:

Planificación: En esta fase inicial hay que constatar la verdadera necesidad del producto que se va a desarrollar y valorar los recursos humanos y técnicos que precisa su desarrollo. Esta valoración se traduce en coste económico y tiempo de elaboración. Si se aprueba el proyecto, se pasa a la fase siguiente.

[7]La organización de estas fases da lugar a diversas variantes del ciclo de vida, siendo uno de los más aplicados el conocido como ciclo de vida *clásico*. Existen críticas importantes a éste, básicamente por su carácter secuencial y por la dificultad de establecer inicialmente todas las especificaciones. En respuesta a estas críticas se han creado otros modelos de desarrollo como son la utilización de prototipos y de lenguajes de cuarta generación (4GL) y el llamado modelo en espiral.

Análisis: En la fase de análisis se establecen cuáles deben ser la funciones que debe cumplir la aplicación y cómo debe realizarse el trabajo conjunto de los diferentes módulos en que se va a dividir. Dentro de esta fase se determina un sistema de pruebas que permita detectar los posibles errores y asegurar el funcionamiento correcto de la aplicación y las condiciones para unir los módulos de forma fiable. Como resultado de esta fase se redactan las especificaciones detalladas del funcionamiento general del *software*.

Diseño: En esta fase se diseña el conjunto de bloques, se dividen en partes y se asignan a los equipos de programadores. Cada equipo elabora su parte, escribiéndola en un lenguaje algorítmico y probándola de forma manual. Como resultado de esta fase se obtienen algoritmos escritos en lenguaje algorítmico.

Codificación: La fase de codificación se confunde muchas veces con la programación y consiste en escribir los algoritmos en un lenguaje de programación. Es un proceso casi automático.

Validación: La fase de validación consiste en aplicar el sistema de pruebas a los módulos, a las conexiones entre ellos (prueba de integración) y, por último, a la totalidad de la aplicación (prueba de validación). Como resultado de esta fase se genera la aplicación habiendo corregido todos los errores detectados en las pruebas.

Mantenimiento: En la fase de mantenimiento se redacta la documentación actualizada, tanto para el programador como para el usuario. Se inicia la explotación de la aplicación y se detectan y corrigen los errores y deficiencias no advertidas en las fases anteriores, lo que puede suponer un coste añadido importante. El resultado de esta fase es una aplicación en explotación.

Evidentemente, en un curso como éste, los programas que vamos a desarrollar no requieren, por su reducido tamaño, la ejecución de todas estas fases. Sin embargo, conviene advertir que la tarea de la programación no consiste en la codificación (escritura de un programa) directa, ni siquiera para programas sencillos, sino que incluye otras fases: es necesario comprender primero el problema que se plantea; si se puede dividir en subproblemas, esto reduce la complejidad, aunque en ese caso hace falta acoplar los diferentes módulos (por el momento, digamos que se trata de fragmentos de programa); y, por supuesto, es ineludible que el programa sea correcto, de forma que este requisito necesario merece toda nuestra atención.

1.6 Conclusión

La importancia de los algoritmos y de los lenguajes algorítmicos reside en su capacidad para expresar procesos, encaminados a ofrecer la solución a problemas planteados, con independencia de los lenguajes de programación, y aun de la máquina, siendo así posible el desarrollo de algoritmos para computadores todavía no creados: de hecho, muchos de los programas actuales responden a algoritmos concebidos mucho antes de que apareciera el primer computador: considérese por ejemplo el método que ideó Euclides en el siglo IV a. C. para hallar el máximo común divisor de dos números naturales. Los lenguajes de programación son también importantes, ya que permiten materializar los algoritmos, razón por la que se ha avanzado tanto en el estudio de éstos en las últimas décadas.

Por otra parte, tratar el método (*algoritmo*) como objeto de estudio supone un salto de nivel en la resolución de problemas, ya que ahora no sólo se requiere la aplicación de un determinado procedimiento de resolución, sino que cobra importancia la invención y descripción del mismo. Es decir, para resolver un problema habrá que identificar qué acciones conducen a la solución, y cómo describirlas y organizarlas.

En otras palabras, en este capítulo, nuestro *modus operandi* ha consistido en:

1. Disponer de un problema y un método.
2. Resolverlo; es decir, aplicar el método al problema.

mientras que, a partir de ahora, sólo disponemos de un problema y no del método que debemos aplicar, por lo que deberemos diseñar y desarrollar el mismo para poder aplicarlo al problema planteado. En resumidas cuentas, la resolución de un problema general requerirá:

1. Diseñar un algoritmo apropiado.
2. Escribir el programa correspondiente en un lenguaje de computador.
3. Ejecutar el programa correspondiente para el juego de datos del problema.

1.7 Ejercicios

1. Considere la relación de problemas:
   (i) Resolver una ecuación de primer grado de la forma $a + bx = 0$.
   (ii) Sumar dos fracciones.
   (iii) Interpretar una partitura al violín.

(iv) Hacer la cuenta atrás, desde 10 hasta 0.

Para cada uno de ellos, se pide que:

(a) Identifique cuáles son los datos y los resultados.

(b) Describa un problema más general y, si se puede, otro menos general.

(c) Distinga cuáles de esos problemas pueden resolverse mediante algoritmos y cuáles no.

(d) Esboce, con sus propias palabras o en seudocódigo, un algoritmo para los problemas (i), (ii) y (iv).

2. El problema de restar dos enteros positivos se puede resolver por un procedimiento análogo al de la suma lenta: en vez de pasar unidades de una cantidad a la otra,

$$(2, 3) \to (1, 4) \to (0, 5) \to 5$$

se van restando unidades a ambas a la vez:

$$(9, 2) \to (8, 1) \to (7, 0) \to 7$$

Se pide que:

(a) Escriba un diagrama de flujo para este problema.

(b) Examine cómo evoluciona para el cálculo de $5 - 2$.

(c) Estudie su complejidad.

(d) Estudie su corrección.

(e) Exprese el algoritmo en seudocódigo.

(f) Redacte el programa correspondiente en Pascal basándose en el programa presentado para la suma lenta.

1.8 Referencias bibliográficas

Muchos de los conceptos que se estudian en este capítulo (y en algunas otras partes de este libro) pueden consultarse también en [GL86], que ofrece una introducción clara y amena a diversos aspectos teóricos y prácticos sobre los algoritmos y programas, su desarrollo y ejecución, así como sus implicaciones sociales. La formalización de los algoritmos dada aquí es una adaptación de la presentada en [FS87].

El apartado de los lenguajes algorítmicos se amplía un poco en el capítulo 7 de este mismo libro, donde se introducen el seudocódigo y los diagramas de flujo. Debe advertirse sin embargo que esta referencia *no* anima al lector a estudiar los diagramas de flujo ahora, sino más bien a esperar su momento y limitar su estudio al cometido con que allí se introducen.

A pesar del título del epígrafe "lenguajes algorítmicos y de programación", no se ha dicho mucho sobre estos últimos: en realidad, la idea es establecer un primer vínculo entre ellos. Para ampliar lo dicho aquí sobre los lenguajes de programación, remitimos al capítulo 5 de [PAO94] y a la bibliografía que allí se refiere.

Capítulo 2

El lenguaje de programación Pascal

En este breve capítulo se presenta el lenguaje de programación Pascal, resaltando algunas de sus características más importantes, como la sencillez de sus instrucciones, la estructuración de éstas y su capacidad expresiva, que permite implementar algoritmos de muy diversa índole. Estas características, entre otras, son las que justifican que Pascal sea el lenguaje de programación utilizado en este libro.

2.1 Introducción

Pascal es un lenguaje *de alto nivel*: se parece más al lenguaje natural hablado, o al matemático, que al lenguaje de máquina (véase el cap. 5 de [PAO94]). Este nivel se alcanza gracias a una pequeña colección de mecanismos simples pero de una gran potencia: unos permiten estructurar acciones (secuencia, selección e iteración), otros datos (arrays, registros, ficheros), y otros hacen posible extender el lenguaje, dotándolo en general con conceptos (datos y operaciones) semejantes a los empleados en el razonamiento humano sobre los problemas, como se mostró en el apartado 1.4.

La gran *expresividad* debida a la particular estructuración de sus datos y de su pequeña colección de instrucciones evita la necesidad de recurrir a otros mecanismos (probablemente enrevesados, difíciles de expresar y de analizar, como por ejemplo la instrucción de bifurcación incondicional **goto**); por todo esto decimos que es un lenguaje *estructurado* (véase el capítulo 7). Por otra parte, permite crear módulos que extienden el lenguaje (por lo que se dice que es *modular*; ver el capítulo 9); esta característica permite desarrollar programas dividiéndolos en partes (módulos o subprogramas) más pequeñas, independientes, que se pueden desarrollar por separado.

Una característica positiva de Pascal es su reducido conjunto de instrucciones, lo que lo hace relativamente compacto y fácil de aprender. En resumen, el lenguaje Pascal facilita la adquisición de buenos hábitos de programación y proporciona los instrumentos que permiten adaptarse a los principales métodos de desarrollo de algoritmos: programación estructurada y modular y definición y uso de tipos de datos. Estas técnicas han convertido en las últimas décadas a la programación en una actividad disciplinada y guiada por una cierta metodología, y el lenguaje Pascal ha contribuido enormemente a su difusión y utilización. Por ello, es además idóneo como primer lenguaje de programación, facilitando el aprendizaje posterior de otros. Así lo prueba su fuerte implantación.

2.2 Otros detalles de interés

La mayoría de los traductores del Pascal son *compiladores* (véase el apartado 5.3.1 de [PAO94]): el programa en Pascal se traduce de una sola vez a lenguaje máquina antes de ser ejecutado, y en ese proceso se detectan gran cantidad de errores de forma automática, permitiendo al programador enmendarlos antes de la ejecución. Como ejemplo de las verificaciones que se efectúan durante la compilación, una de las más importantes consiste en la compatibilidad de los tipos de los objetos.

Antes de la aparición de Pascal existían lenguajes dirigidos a la programación científica (como Fortran) y otros dirigidos a la de gestión (como Cobol). El lenguaje Pascal trata de conciliar los dos tipos de programación, por lo que suele decirse que Pascal es un lenguaje *de propósito general*.

2.3 Origen y evolución del lenguaje Pascal

Es obvio decir que Pascal toma el nombre del matemático francés Blaise Pascal (1623–1662) que en 1642 inventó la primera máquina de calcular para ayudar a su padre en su trabajo de tasador de impuestos.

El lenguaje Pascal fue concebido por Niklaus Wirth en 1968 y definido en 1970 en el Instituto Politécnico de Zurich para enseñar la programación a sus alumnos. Desde que comenzó a utilizarse (1971), ha tenido un enorme desarrollo y difusión, adaptándose a la mayoría de los computadores, grandes y pequeños. Actualmente es uno de los lenguajes más usados en las universidades de muchos países del mundo. Gracias a esta difusión, junto con los compiladores de este lenguaje, se han desarrollado potentes entornos de programación de gran calidad y bajo precio.

Algunas de las implementaciones del lenguaje son Turbo Pascal© (que funciona en computadores compatibles PC, bajo el sistema operativo DOS y bajo Windows), Macintosh Pascal©, VAX Pascal©, Microsoft Pascal© y Quick Pascal©.

Es un lenguaje estandarizado, estando recogido en el *Pascal User Manual and Report* de K. Jensen y N. Wirth [JW85]. Por lo general, las distintas versiones se adaptan al estándar y lo extienden. Por lo tanto, un programa escrito en Pascal estándar (según el *Pascal User Manual and Report*) debe funcionar en la mayoría de las versiones; en cambio, si una versión contiene extensiones, lo más probable es que no funcione en las otras.

En cualquier caso, es ciertamente comprensible que las características presentadas aquí, sin conocer el lenguaje, pueden sonar a hueco, ya que el momento apropiado para una valoración cabal es *a posteriori*, después de un conocimiento más completo de este lenguaje e incluso otros: sólo así puede apreciarse su elegancia conceptual, la enorme influencia que ha tenido en el desarrollo de otros, en la enseñanza de la programación y en la metodología de desarrollo de programas y, naturalmente, también sus limitaciones.

Quienes ya conozcan este lenguaje en mayor o menor medida, o quienes deseen ampliar el contenido de este libro, pueden encontrar en [Wir93] una visión panorámica, escrita por el propio Wirth.

2.4 Pascal y Turbo Pascal

La posición que se adopta en este libro acerca del lenguaje de programación utilizado es intermedia, entre el estándar ideado inicialmente (lo que es conveniente para que los programas sean transportables) y un compilador real (lo que tiene la ventaja de permitir la práctica en el desarrollo de programas). El compilador concreto adoptado es Turbo Pascal, potente, rápido, de amplia difusión y dotado con un entorno muy bien desarrollado que facilita la tarea del programador.

El inconveniente consiste en que Turbo Pascal, como la mayoría de los compiladores existentes, incluye diferencias con respecto a Pascal estándar, aunque

éstas son mínimas: debe decirse que casi no tiene limitaciones, y que en cambio ofrece una gran cantidad de extensiones.

Nuestra decisión ha sido trabajar con este compilador[1] para que sea posible al alumno desarrollar sus prácticas, haciendo uso de las mínimas herramientas extendidas del mismo y señalando en todo caso las diferencias con respecto al estándar. Además se incluye un apéndice sobre Turbo Pascal donde se presentan las características y diferencias más importantes con Pascal estándar.

[1]Para nuestros propósitos es válida cualquier versión de Turbo Pascal desde la 4.0 hasta la 7.0, así como Turbo Pascal para Windows y Borland Pascal.

Capítulo 3

Tipos de datos básicos

Es preciso adquirir desde el principio el concepto de dato, puesto que los algoritmos (y por lo tanto los programas) los manejan constantemente,[1] desde la entrada (que puede considerarse como la colección inicial de datos) hasta la salida (o conjunto de datos resultantes), pasando frecuentemente por un sinfín de resultados provisionales que intervienen en los cálculos intermedios (tales como las cifras "de acarreo" en una operación de multiplicar).

Este capítulo se dedica al estudio de los tipos de datos elementales de Pascal, explicando sus dominios y operaciones, así como la construcción de expresiones con datos de dichos tipos.

[1] De hecho, en un algoritmo se organizan dos aspectos inseparables: acciones y datos.

3.1 Introducción

Un *dato* es simplemente la representación de un objeto mediante símbolos manejables por el computador.

El uso de números (enteros y reales), caracteres y valores lógicos es corriente en programación. De hecho, casi todos los lenguajes disponen de ellos *a priori*, por lo que se les llama también *tipos de datos predefinidos* o también *estándar*.

Ciertamente, es posible ampliar nuestro lenguaje con otros objetos a la medida de los problemas que se planteen, tales como vectores o conjuntos (véanse los capítulos 11 y 12). Éstos se construyen usando los datos predefinidos como las piezas más elementales, por lo que también se les llama tipos de datos *básicos* a los tipos de datos predefinidos.

Sin embargo, lo cierto es que, en principio, sólo se puede contar con los tipos de datos mencionados, que en Pascal se llaman respectivamente `integer`, `real`, `char` y `boolean`, y que estudiamos a continuación.[2]

Los tipos de datos se caracterizan mediante sus *dominios* (conjuntos de valores) y las *operaciones* y *funciones* definidas sobre esos dominios.

3.2 El tipo integer

Dominio

Su *dominio*, denotado $\mathcal{Z}$, incluye valores enteros positivos o negativos. Debido a las limitaciones de representación (véase el apartado 2.2.3 de [PAO94]), el dominio de `integer` está acotado por la constante predefinida `MaxInt`, propia de cada versión: en Turbo Pascal por ejemplo, `MaxInt` vale `32767`, y el dominio de `integer` es {`-32768,..., 32767`}. Los números `integer` se escriben literalmente sin espacios ni puntos entre sus cifras, posiblemente precedidos por el signo más o menos, por ejemplo: `174`, `-273`, `+17`, etc. El diagrama sintáctico (véase apartado 5.2.1 de [PAO94]) de la figura 3.1 resume estos detalles.

Operaciones y funciones

Asociadas al tipo `integer`, se tienen las siguientes operaciones aritméticas:

[2]En adelante, usaremos `letra de molde` para escribir el texto en Pascal, excepto las palabras reservadas (véase el apartado 4.2) que aparecerán en **negrilla**.

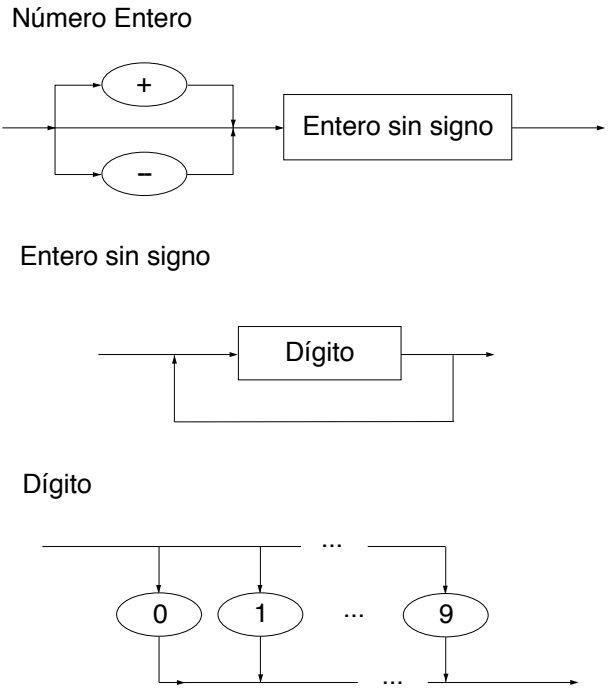

Número Entero
+
–
Entero sin signo
Entero sin signo
Dígito
Dígito
...
0
1
...
9
...

Figura 3.1.

`+`	suma
`-`	resta
`*`	multiplicación
div	división entera
mod	resto de la división entera

Su funcionamiento es similar al que tienen en Matemáticas: son operaciones *binarias* (tienen dos argumentos, salvo la operación `-`, usada también para hallar el opuesto de un entero) e *infijas* (en las expresiones, se sitúan entre sus dos argumentos). Emplearemos la notación matemática

$$\mathbf{div} : \mathcal{Z} \times \mathcal{Z} \longrightarrow \mathcal{Z}$$

para expresar que los dos argumentos son de tipo entero y su resultado también lo es. Todas las operaciones presentadas responden a este esquema. Las operaciones **div** y **mod** expresan respectivamente el cociente y el resto de la división entera:[3]

`7`	**div**	`2`	$\rightsquigarrow$	`3`	`7`	**mod**	`2`	$\rightsquigarrow$	`1`
`7`	**div**	`-2`	$\rightsquigarrow$	`-3`	`7`	**mod**	-2	$\rightsquigarrow$	`1`
`-7`	**div**	`2`	$\rightsquigarrow$	`-3`	-7	**mod**	`2`	$\rightsquigarrow$	-1
`-7`	**div**	`-2`	$\rightsquigarrow$	`3`	-7	**mod**	-2	$\rightsquigarrow$	-1

Obsérvese que verifican la conocida regla:

$$\mathit{dividendo} = \mathit{divisor} * \mathit{cociente} + \mathit{resto}$$

Además, existen las siguientes funciones predefinidas:

`Abs`	valor absoluto del entero
`Sqr`	cuadrado del entero
`Pred`	entero predecesor
`Succ`	entero sucesor

que son *monarias* (se aplican a un argumento único) y se aplican en forma *prefija* (preceden a su argumento, dado entre paréntesis).

Estas operaciones y funciones son *internas*, puesto que convierten argumentos `integer` en `integer`:

$$\texttt{Abs} : \mathcal{Z} \longrightarrow \mathcal{Z}$$

Sin embargo, debe observarse que las limitaciones que pesan sobre el tipo `integer` afectan a las operaciones y funciones, cuyo resultado será correcto sólo cuando no rebase los límites del dominio.

[3] Usaremos el símbolo $\rightsquigarrow$ para expresar la evaluación de una operación o función.

Expresiones

Al igual que ocurre en Matemáticas, es posible escribir expresiones combinando los números con esas operaciones y funciones:

```
2 + 3 * Sqr(2)
```

Entonces, tiene sentido preguntarse en qué orden se reducirán las operaciones involucradas, ya que el resultado puede depender de ello. Para deshacer la ambigüedad se conviene en evaluar las operaciones `*`, **div** y **mod** antes que `+` y `-`, como se hace en Matemáticas; esta idea se expresa habitualmente diciendo que la precedencia de unas operaciones (`*`, **div** y **mod**) es mayor que la de otras. Por ejemplo, la expresión anterior se calcularía en tres pasos:

`2 + 3 * Sqr(2)` $\rightsquigarrow$ `2 + 3 * 4` $\rightsquigarrow$ `2 + 12` $\rightsquigarrow$ `14`

Similarmente, una expresión puede resultar ambigua cuando sea posible aplicar dos operaciones con la misma precedencia. Por ejemplo:

`7` **div** `2 * 2`

En este caso, se efectuará la valoración de las operaciones asumiendo la asociatividad a la izquierda:

`7` **div** `2 * 2` $\rightsquigarrow$ `3 * 2` $\rightsquigarrow$ `6`

Como ocurre en Matemáticas, es posible forzar un orden de evaluación distinto del convenido mediante el uso de paréntesis:

`(2 + 3) * 4` $\rightsquigarrow$ `5 * 4` $\rightsquigarrow$ `20`

`7` **div** `(2 * 2)` $\rightsquigarrow$ `7` **div** `4` $\rightsquigarrow$ `1`

Aunque las operaciones anteriores resultan sencillas, combinándolas apropiadamente es posible lograr expresiones que resuelven directamente un gran número de problemas. Por ejemplo, si `n` representa una cierta cantidad (entera) de dinero, en pesetas, `n` **div** `25` expresaría su cambio en monedas de cinco duros; `n` **mod** `25` el número de pesetas restantes, que se podrían cambiar por `n` **mod** `25` **div** `5` duros y `n` **mod** `5` pesetas.

Con el uso de las funciones, las expresiones adquieren una gran potencia. Como ejemplo, la expresión

```
(a + b + 1) div 2 + Abs(a - b) div 2
```

calcula el máximo de dos cantidades enteras representadas por `a` y `b`.

Por otra parte, el orden de aplicación de las funciones está exento de ambigüedad debido al uso obligatorio de los paréntesis.

3.3 El tipo real

Dominio

Su dominio, en adelante $\mathcal{R}$, incluye valores numéricos con parte decimal. Las limitaciones en la representación de cantidades reales afectan ahora al dominio en dos aspectos (véase el apartado 2.2.3 de [PAO94]): la magnitud de los valores incluidos (en Turbo Pascal por ejemplo, el dominio de `real` está comprendido aproximadamente entre $\pm 1.7 * 10^{38}$) y la precisión (hacia 10 cifras significativas en Turbo Pascal).

Hay dos modos de escribir los números reales en Pascal: usando el punto decimal (que debe tener algún dígito a su izquierda y a su derecha), o bien usando la llamada *notación científica* o *exponencial*. A continuación tenemos ejemplos de números reales en notación decimal y científica, respectivamente:

```
3.1415926    6.023E+23
```

El diagrama sintáctico de la expresión de números reales es el de la figura 3.2.

Operaciones y funciones

Existen las siguientes operaciones aritméticas relacionadas con el tipo `real`:

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

y las siguientes funciones predefinidas:

`Abs`	valor absoluto
`Sqr`	cuadrado
`SqRt`	raíz cuadrada
`Sin`	seno
`Cos`	coseno
`ArcTan`	arcotangente
`Ln`	logaritmo neperiano
`Exp`	función exponencial

En las funciones trigonométricas, el ángulo se expresa en radianes. Las funciones logarítmica y exponencial son de base e.

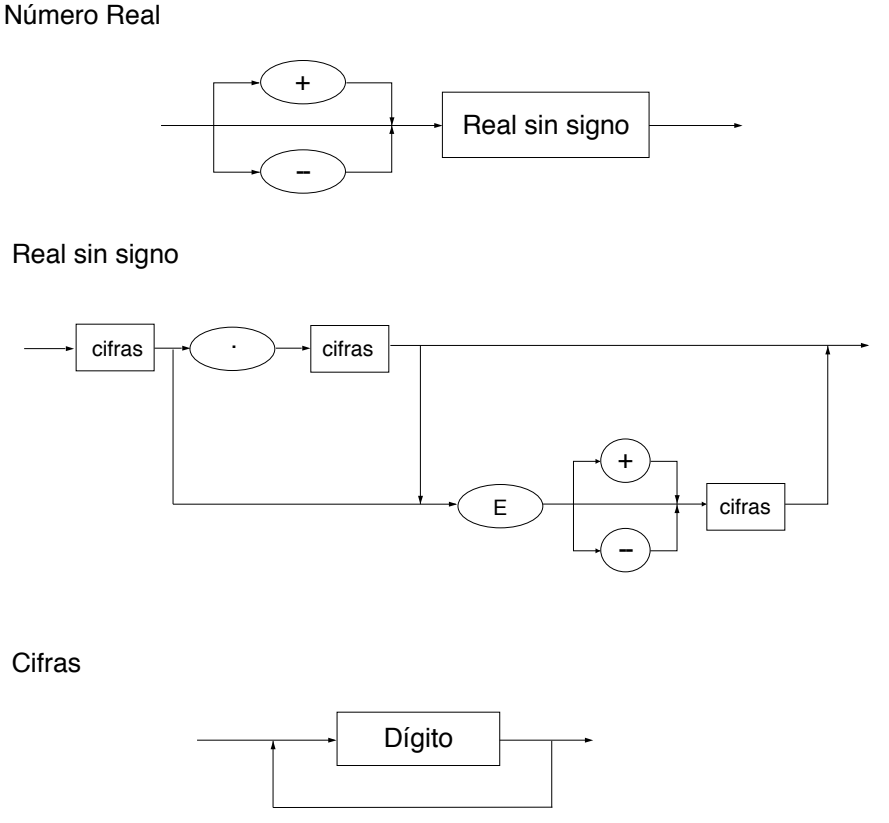

Número Real
+
−
Real sin signo
Real sin signo
cifras
.
cifras
E
+
−
cifras
Cifras
Dígito

Figura 3.2.

Reales y enteros

Se observa que los símbolos de las operaciones `+`, `-`, y `*`, así como las funciones `Abs` y `Sqr`, coinciden con las correspondientes a los números enteros, por lo que las siguientes descripciones resultan ambas correctas,

$$\texttt{+} : \mathcal{Z} \times \mathcal{Z} \longrightarrow \mathcal{Z}$$

$$\texttt{+} : \mathcal{R} \times \mathcal{R} \longrightarrow \mathcal{R}$$

así como las siguientes para el caso de las funciones:

$$\texttt{Abs} : \mathcal{Z} \longrightarrow \mathcal{Z}$$

$$\texttt{Abs} : \mathcal{R} \longrightarrow \mathcal{R}$$

Este multiuso de ciertas operaciones y funciones se llama *sobrecarga* de operadores, y de él aparecerán nuevos casos más adelante.

Otro aspecto en que el lenguaje se muestra flexible consiste en reconocer la inclusión $\mathcal{Z} \subset \mathcal{R}$, y obrar en consecuencia: así, toda operación o función que requiera expresamente una cantidad `real` aceptará una de tipo entero, realizando automáticamente la correspondiente conversión. La descripción siguiente

$$\texttt{+} : \mathcal{Z} \times \mathcal{R} \longrightarrow \mathcal{R}$$

es una consecuencia de ello, producida al convertir el primer argumento en un `real`. En cambio, la conversión en sentido inverso no se realiza automáticamente. Para ello, el lenguaje está dotado con funciones para convertir reales en enteros:

`Trunc`	truncamiento (eliminación de la parte decimal)
`Round`	redondeo al entero más próximo[4]

Por ejemplo,

```
Round(-3.6)   ⇝   -4
Trunc(-99.9)  ⇝   -99
-Round(99.9)  ⇝   -100
-Round(-99.9) ⇝   100
```

[4]Inferior o superior.

Expresiones

La precedencia de las operaciones *aditivas* (`+` y `-`) es menor que la de las *multiplicativas* (`*` y `/`), igual que en Matemáticas.

El juego de operaciones y funciones predefinidas en Pascal tiene carencias notables, tales como la potencia, la tangente, el arcoseno o los logaritmos de base decimal o arbitraria. Sin embargo, combinando las presentadas anteriormente, se logra fácilmente expresar las operaciones mencionadas, entre otras muchas. Por ejemplo, en la siguiente tabla se dan algunas expresiones equivalentes (salvo errores de precisión)[5] a algunas funciones usuales:

función	expresión equivalente	
x^y	`Exp(y * Ln(x))`	para $x > 0$
$tg(x)$	`Sin(x)/Cos(x)`	para $x \neq \frac{\pi}{2} + k\pi$
$arcsen(x)$	`ArcTan(x/SqRt(1 - Sqr(x))`	para $0 < x < 1$
$log_b(x)$	`Ln(x)/Ln(b)`	para $b > 1, x > 0$

Como ejemplo final de la potencia de las operaciones y funciones presentadas, damos la expresión

```
Trunc (Ln(n)/Ln(10)) + 1
```

que halla el número de cifras de un entero $n > 0$.

3.4 El tipo char

Dominio

El dominio de este tipo de datos, abreviadamente $\mathcal{C}$, incluye el juego de caracteres disponibles en el computador, que varía entre las diferentes versiones desarrolladas, todas tienen en común el respeto al orden alfabético inglés y al orden entre dígitos; por ejemplo, en Turbo Pascal se utiliza una codificación en ASCII de 8 bits, por lo que existen hasta 256 posibles caracteres que se recogen en la correspondiente tabla de códigos (véase el apartado 2.2.4 de [PAO94]).

Los valores de tipo `char` se escriben entre apóstrofes (por ejemplo, `'A'`) excepto el carácter apóstrofe (`'`), que se escribe mediante `''''`.

[5]Debe tenerse en cuenta que la imprecisión cometida al tratar con números reales puede hacerse más importante a medida que se suceden las operaciones.

Funciones

No existen operaciones internas entre caracteres; en cambio, sí se tienen las siguientes funciones internas predefinidas:

`Pred`	carácter anterior
`Succ`	carácter sucesor

Además, existen las siguientes funciones de conversión entre `char` e `integer`:

`Ord`	número de orden del carácter en el juego adoptado
`Chr`	carácter asociado a un número de orden dado

El esquema de estas funciones es el siguiente:

$$\begin{array}{rcccc} \texttt{Pred, Succ} & : & \mathcal{C} & \longrightarrow & \mathcal{C} \\ \texttt{Ord} & : & \mathcal{C} & \longrightarrow & \mathcal{Z} \\ \texttt{Chr} & : & \mathcal{Z} & \longrightarrow & \mathcal{C} \end{array}$$

Por ejemplo,

```
          Pred('Z')  ⇝  'Y'
          Pred('z')  ⇝  'y'
          Succ('7')  ⇝  '8'
Chr(Ord('1') + 4)    ⇝  '5'
```

3.5 El tipo boolean

Dominio

Este tipo proviene del álgebra de Boole, como indica su nombre. Su dominio, denotado mediante $\mathcal{B}$ en adelante, contiene solamente los dos valores lógicos predefinidos: `False` y `True`. El tipo `boolean` es particularmente útil en tareas de control de bucles y selecciones.

Operaciones y funciones internas

Asociadas al tipo `boolean`, se tienen las siguientes operaciones:

not	negación lógica (con la precedencia más alta)
and	conjunción lógica (precedencia multiplicativa)
or	disyunción lógica (predecencia aditiva)

Son internas (convierten valores lógicos en valores lógicos):

$$\begin{array}{lcl} \textbf{not} & : & \mathcal{B} \longrightarrow \mathcal{B} \\ \textbf{and} & : & \mathcal{B} \times \mathcal{B} \longrightarrow \mathcal{B} \\ \textbf{or} & : & \mathcal{B} \times \mathcal{B} \longrightarrow \mathcal{B} \end{array}$$

y su funcionamiento viene dado por la siguiente tabla de verdad:

A	B	A **and** B	A **or** B	**not** A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

Además, se tienen predefinidas las funciones sucesor, predecesor y orden, que operan como sigue:

$$\begin{array}{c} \texttt{Succ(False)} \rightsquigarrow \texttt{True} \\ \texttt{Pred(True)} \rightsquigarrow \texttt{False} \\ \texttt{Ord(False)} \rightsquigarrow \texttt{0} \\ \texttt{Ord(True)} \rightsquigarrow \texttt{1} \end{array}$$

Operadores relacionales

Son las operaciones binarias de comparación siguientes:

`=`	igual
`<>`	distinto
`<`	menor
`<=`	menor o igual
`>`	mayor
`>=`	mayor o igual

Estas operaciones están *sobrecargadas*: permiten la comparación entre dos valores de cualquiera de los tipos básicos, resultando de ello un valor lógico: Es decir, si representamos los cuatro tipos introducidos mediante $\mathcal{Z}$, $\mathcal{R}$, $\mathcal{C}$ y $\mathcal{B}$ respectivamente, se tiene

$$\begin{array}{lcl} \texttt{>=} & : & \mathcal{Z} \times \mathcal{Z} \longrightarrow \mathcal{B} \\ \texttt{>=} & : & \mathcal{R} \times \mathcal{R} \longrightarrow \mathcal{B} \\ \texttt{>=} & : & \mathcal{C} \times \mathcal{C} \longrightarrow \mathcal{B} \\ \texttt{>=} & : & \mathcal{B} \times \mathcal{B} \longrightarrow \mathcal{B} \end{array}$$

para cada una de las operaciones relacionales.

Con respecto a la comparación de números reales, debemos remitirnos al apartado 2.2.3 de [PAO94], de donde se deduce la escasa fiabilidad de estas operaciones cuando los argumentos están muy próximos, como consecuencia de las limitaciones de precisión en los sistemas de representación de estos números.

En cuanto a la comparación de datos no numéricos, el símbolo `<` significa "anterior" y no "menor" que, obviamente, no tiene sentido en tales casos. En el caso concreto de `char`, esta anterioridad viene dada por sus posiciones en la tabla adoptada (v.g. ASCII), y en el de `boolean`, se considera `False` anterior a `True`.

Algunas operaciones relacionales reciben nombres especiales cuando trabajan sobre valores booleanos:

`=`	equivalencia lógica
`<>`	o exclusivo
`<=`	implicación

El funcionamiento de estas operaciones viene dado por las siguientes tablas de verdad:

A	B	A = B	A <> B	A <= B
False	False	True	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	False	True

Las operaciones relacionales tienen menor precedencia que las aditivas. Por ejemplo:

```
    3 + 1 >= 7 - 2 * 3
~>  3 + 1 >= 7 - 6        {ops. multiplicativos}
~>  4 >= 1                {ops. aditivos}
~>  False                 {ops. relacionales}
```

Finalmente, se tiene la función de conversión que indica si un entero es impar:

$$\texttt{Odd} : \mathcal{Z} \longrightarrow \mathcal{B}$$

$$\texttt{Odd(n)} \rightsquigarrow \begin{cases} \texttt{True} & \text{si } n \text{ es impar} \\ \texttt{False} & \text{en otro caso} \end{cases}$$

Aunque la función `Odd(n)` es equivalente a `n` **mod** `2 = 1`, su uso es preferible al de ésta, ya que opera más eficientemente.

Circuito largo-corto

Sea la expresión booleana `P` **and** `Q`. Un modo de valorarla consiste en hallar primero el valor de `P`, luego el valor de `Q`, y entonces deducir el valor de la expresión de las tablas de la verdad. No obstante, si `P` hubiera resultado valer `False`, el valor de la expresión será `False`, tanto si `Q` resulta ser `True` como si resulta ser `False`, de manera que podríamos haber evitado hallar `Q`. Estos dos modos de evaluar las expresiones booleanas se llaman respectivamente con *circuito largo* y con *circuito corto*.

Su importancia no radica únicamente en la eficiencia conseguida al evitar evaluar ciertos términos, sino que a veces el modo de evaluación tiene fines algo más sutiles. Consideremos la valoración de la expresión siguiente, donde `den` resulta valer 0:

```
(den <> 0) and (num/den = 0)
```

Si el modo de evaluación es con circuito corto, el resultado final es `False`, para lo cual sólo se necesita hallar el valor de

```
den <> 0
```

En cambio, si se valora con circuito largo, el cálculo del segundo término produce un error ("división por cero"), y la ejecución del programa fracasa. Pascal estándar establece que la evaluación se produce con circuito largo, pero Turbo Pascal ofrece la posibilidad de escoger entre ambos modos (véase el apartado C.3).

3.6 Observaciones

Tipos ordinales

En los cuatro tipos de datos simples predefinidos existe una ordenación, de manera que todos ellos son comparables con los operadores relacionales. Por otra parte, es posible enumerar fácilmente los dominios $\mathcal{Z}$, $\mathcal{C}$ y $\mathcal{B}$, asignando a sus elementos su número de posición: ése es el cometido de la función `Ord`, que está definida para estos tres tipos de datos, y no para $\mathcal{R}$.

Por ello, decimos que los tipos `integer`, `char` y `boolean` son *ordinales*, mientras que `real` no lo es. Además de `Ord`, las funciones `Succ` y `Pred` son exclusivas de los tipos ordinales, por lo que no es posible ni razonable obtener el siguiente (o el anterior) de un número real dado.

Resumen de expresiones con los tipos de datos básicos

Las expresiones en programación son similares a las empleadas en Matemáticas, aunque sus elementos constituyentes no sólo son números, como hemos visto. Las expresiones pueden consistir en una constante, una variable, una función aplicada a una expresión o una operación entre expresiones. Aunque en principio puede resultar extraño definir expresiones en términos de expresiones, debe observarse que estas últimas son componentes (esto es, más pequeñas), con lo que finalmente, una expresión está compuesta por constantes o variables, piezas básicas de las expresiones. En las figuras 3.3 y 3.4 se completa lo explicado hasta ahora sobre la estructura sintáctica que tienen las expresiones válidas[6] en Pascal.

Como se indicó anteriormente, las operaciones se aplican en las expresiones por orden, según su precedencia, como se indica en la tabla de la figura 3.5. Si coinciden en una expresión dos o más operaciones de la misma precedencia se asocian de izquierda a derecha. El orden de aplicación de precedencias puede alterarse mediante el paréntesis, igual que en Matemáticas.

Las funciones se aplican a sus argumentos entre paréntesis, por lo que no existe ambigüedad posible.

Cadenas de caracteres

Si se observa la sintaxis de los literales, se ve que una posibilidad consiste en escribir una cadena de ningún carácter, uno o más. Aunque este literal no pertenece a ninguno de los tipos de datos básicos presentados, esta posibilidad nos permitirá desde el principio emitir mensajes claros (véase 4.3.2).

Es válido por tanto escribir literales como los siguientes:

```
'Yo tengo un tío en América'
''
'Carlos O''Donnell'
```

para representar las frases encerradas entre apóstrofes, sin éstos, que actúan sólo como delimitadores, y expresando el apóstrofe simple mediante dos de ellos. Así por ejemplo, los literales anteriores representan respectivamente la frase `Yo tengo un tío en América`, la cadena vacía y `Carlos O'Donnell`.

[6]En realidad, se consideran sólo las expresiones válidas "por el momento". A medida que se introduzcan posibilidades nuevas, se completarán los diagramas sintácticos.

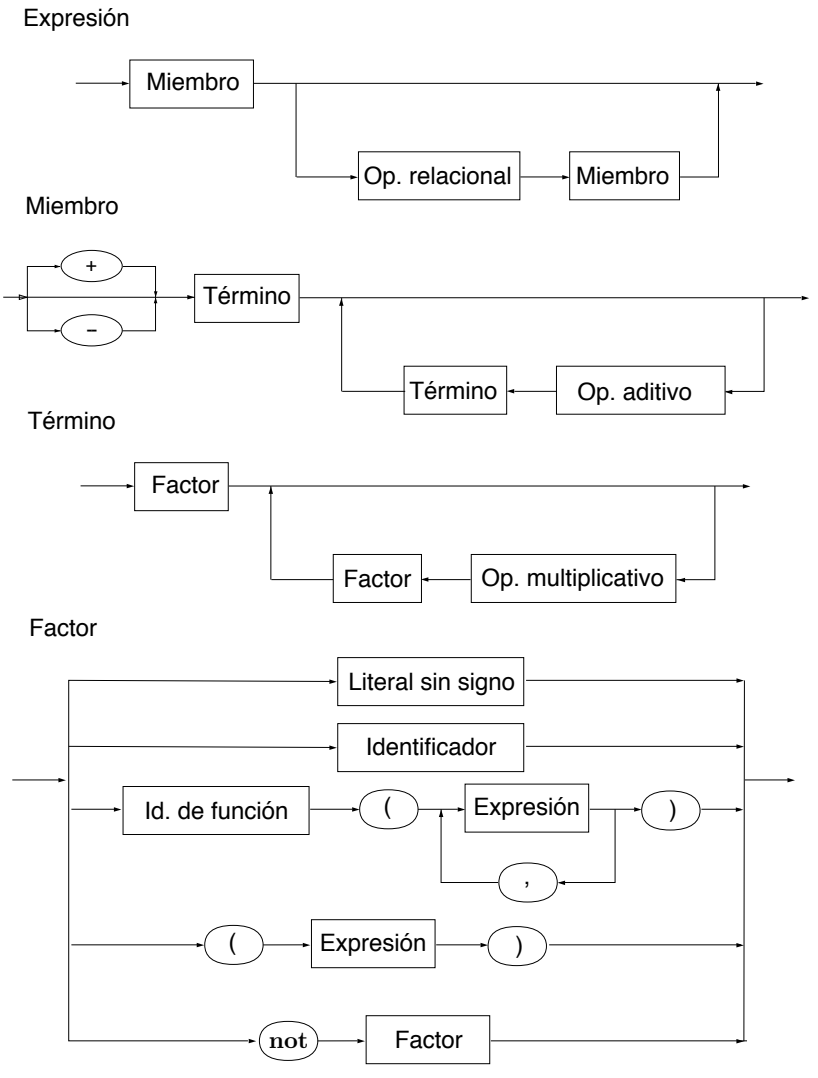


Figura 3.3. Diagrama sintáctico de las expresiones (1).

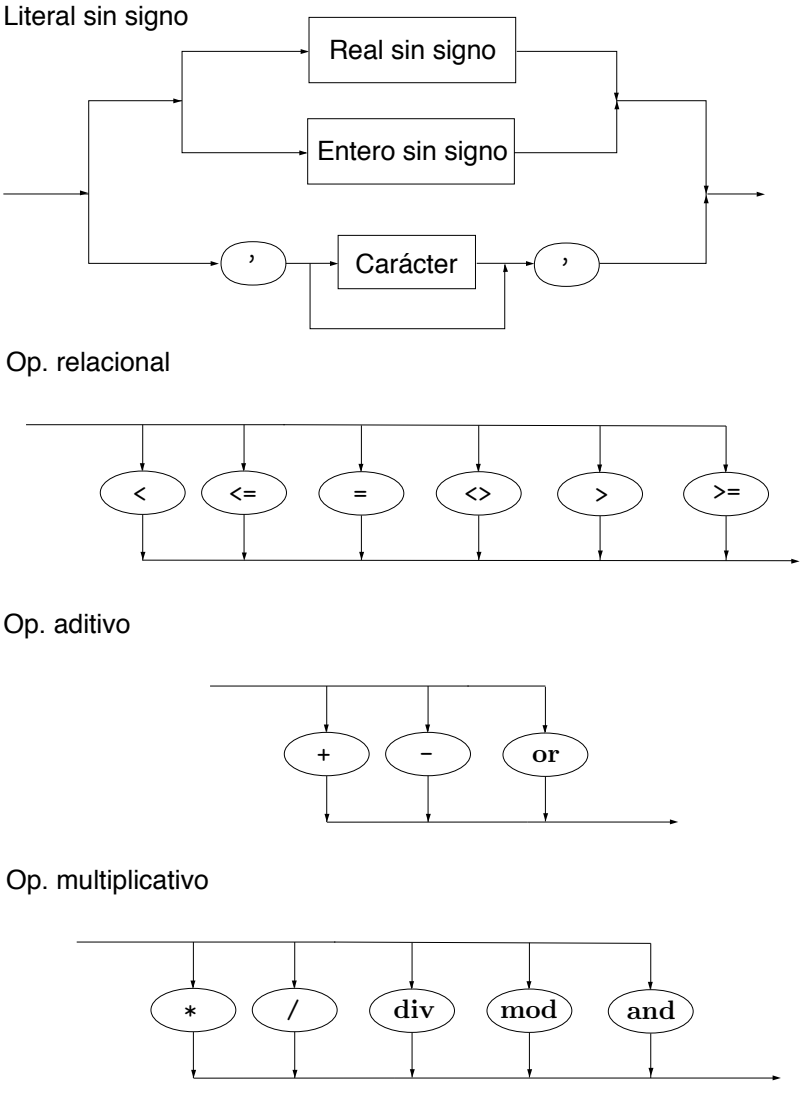


Figura 3.4. Diagrama sintáctico de las expresiones (2).

op. monarios	cambio de signo, **not**
op. multiplicativos	*, /, **div**, **mod**, **and**
op. aditivos	+, -, **or**
ops. de relación	=, <>, <, >, <=, >=

Figura 3.5. Cuadro-resumen de prioridades.

3.7 El tipo de una expresión

El resultado de cada expresión tiene un tipo determinado, independiente de su valor, y que puede conocerse aun ignorando los valores de sus elementos componentes, teniendo en cuenta sólo sus tipos. Así por ejemplo, pueden existir expresiones numéricas, bien enteras (como `2 + 3`) o reales (como `3.14 * Sqr(2.5)` por ejemplo), booleanas (como `2 + 2 = 5`) y de caracteres (como `Succ('a')`).

La información sobre el tipo de una expresión (y el de sus subexpresiones componentes) juega en Pascal un importante papel: una vez escrito un programa, el compilador comprueba que las expresiones que intervienen en él son correctas sintácticamente y que los tipos de las componentes de las mismas son consistentes. Por ejemplo, la expresión siguiente podría ser analizada así:

```
(  (6  +  8)  *  3.14  <  Ord (  'a'  )  )  or  True
(  (Z  +  Z)  *   R    <  Ord (   C   )  )  or   B
(      Z      *   R    <        Z        )  or   B
(         R             <        Z        )  or   B
                  B                        or   B
                        B
```

aceptándose la comparación siguiente:

```
True = ((6 + 8) * 3.14 < Asc('a')) or False
```

y rechazándose en cambio esta otra:

```
2 = ((6 + 8) * 3.14 < Asc('a')) or True
```

3.8 Ejercicios

1. De las siguientes expresiones en Pascal, detecte las erróneas; en las que son correctas, indique qué tipo deben tener los identificadores, y deduzca su tipo y su valor resultante, si es posible.

   (a) `x = y`
   (b) `Odd(k) or Odd(Succ(k))`
   (c) `p = True`
   (d) `10 div 3 = 10 / 3`
   (e) `p > Succ(p)`
   (f) `P = Q or R`
   (g) `Odd(n * (n - 1))`
   (h) `Ord('b') - Ord('a') > 0`

2. Sea α un ángulo, dado en grados. Escriba una expresión en Pascal que halle:

   (a) el número de vueltas completas que da,

   (b) el ángulo entre 0 y 359 al que equivale,

(c) el número del cuadrante en que se encuentra (numerados éstos en sentido inverso al de las agujas del reloj),

(d) el ángulo en radianes al que equivale,

(e) su tangente.

3. Evalúe las siguientes expresiones en el dominio de `integer`:

```
(2 * 2) div 4 = 2 * (2 div 4)
(10000 * 4) div 4 = 10000 * (4 div 4)
```

4. Usando el operador **mod**, escriba en Pascal una expresión que, para un entero `n`, dé el número correspondiente al día de la semana; esto es, para `n = 1`, `2`, ..., `7`, `8`, `9`, ... esa expresión resulte valer `1`, `2`, ..., `7`, `1`, `2`, ...

5. Halle el valor de la expresión `Ord(C) - Ord('0')`, donde `C` es de tipo `char` y representa, sucesivamente, los valores `'0'`, `'1'`, ..., `'9'`.

6. Considere la correspondencia siguiente entre los caracteres alfabéticos y los números naturales:

```
'A'  ⟶  1
...
'Z'  ⟶  26
```

Dé expresiones que pasen del carácter `C`, supuesto que es una mayúscula, al número correspondiente, y del número `N`, supuestamente entre `1` y `26`, a su carácter asociado.

7. Exprese la condición que deben cumplir las coordenadas de un punto del plano (x, y) para:

(a) que su distancia al punto $(1, 1)$ sea inferior a 5 unidades,

(b) estar en el primer cuadrante del plano,

(c) estar por debajo de la recta $x + y = 6$,

(d) cumplir simultáneamente los apartados anteriores.

8. Dados los catetos $c1$ y $c2$ de un triángulo rectángulo, escriba una expresión que sirva para hallar la correspondiente hipotenusa.

9. Se sabe que la relación entre la temperatura, expresada en grados Farenheith (F) y centígrados (C) viene expresada por la fórmula $F = 1'8C + 32$.

(a) Escriba una expresión que sirva para calcular F a partir de C. Deducir el tipo de la misma suponiendo primero que `C` es `integer` y luego que es `real`.

(b) Escriba una expresión que sirva para calcular C a partir de F ofreciendo un resultado `integer`.

10. Encuentre una expresión en Pascal para cada uno de los siguientes apartados:

(a) 10^x, para $x \in \mathbb{R}$

(b) $\log_{10}(x)$, para $x \in \mathbb{R}, x > 0$

(c) $(-1)^n$, para $n \in \mathbb{Z}$

(d) x^y, para $x, y \in \mathbb{R}$

(e) $\sqrt[5]{\frac{1}{n}}$, para $n \in \mathbb{R}$

(f) Dado un entero n, quitarle las c últimas cifras

(g) Dado un entero n, dejar de él sólo sus últimas c cifras

(h) Dado un entero n, hallar la cifra c-ésima

11. Sea $f : \mathbb{R} \longrightarrow \mathbb{R}$ una función derivable. La expresión

$$\frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

proporciona una aproximación de la derivada de f en x, para valores pequeños de ε.

Tomando $\varepsilon = 10^{-3}$, escriba expresiones en Pascal que aproximen las derivadas de las siguientes funciones

(a) $f(x) = \text{sen}(x)$, en $x = \frac{\pi}{3}$

(b) $g(x) = 2x^2 + 3x - 4$, en $x = e$

Capítulo 4

Elementos básicos del lenguaje

En este capítulo se presentan los rudimentos mínimos necesarios para poder construir programas elementales en Pascal. El lector debe esforzarse aquí por tener al término del mismo una visión global de las partes de un programa, así como del modo en que los programas captan los datos, los manipulan y, finalmente, dan los resultados.

4.1 Un ejemplo introductorio

Consideremos el problema de hallar el área de un círculo a partir de su radio. Un procedimiento sencillo para ello consiste en aplicar la conocida fórmula $A = \pi \cdot r^2$, donde A y r son dos números reales que representan el área y el radio del círculo respectivamente. Para llevar a cabo el procedimiento, necesitaremos conocer el valor de la constante π, a la que llamaremos *pi* (si no se necesita una gran precisión, puede bastar con 3.14). Entonces, el cálculo consiste en averiguar el valor del radio r y aplicar la fórmula para hallar A, que se ofrece como el resultado.

Estas ideas pueden organizarse como se indica a continuación:

Manera de hallar el área de un círculo:
  Sean:
    $pi = 3.14$
    $r, A \in \mathbb{R}$ *(radio y área resp.)*
  Pasos que hay que llevar a cabo:
    Averiguar el valor del radio r
    Hallar el valor del área A, que es $pi \cdot r^2$
    El área resulta ser el valor de A

La transcripción de este algoritmo en un programa en Pascal es casi directa:

```
Program AreaCirculo (input, output);
  {Se halla el área de un círculo conociendo su radio}
  const
    Pi = 3.14;
  var
    r, A: real; {radio y área}
begin
  Write('Cuál es el radio?:  ');
  ReadLn(r);
  A:= Pi * Sqr(r);
  WriteLn('Área = ', A)
end. {AreaCirculo}
```

4.2 Vocabulario básico

En castellano las letras se agrupan para formar palabras, y éstas se combinan entre sí y con los signos de puntuación para construir frases; análogamente, en Pascal, se parte de un juego de caracteres básico (ASCII por ejemplo) para componer los diferentes elementos de su vocabulario: las palabras reservadas, los identificadores, los símbolos especiales, los literales y los comentarios.

Palabras reservadas

Las *palabras reservadas* son componentes con significado fijo usadas en los constructores del lenguaje. Se suelen escribir en negrita, facilitando así la lectura de los programas. Las palabras reservadas de Pascal estándar son las siguientes:

and, **array**, **begin**, **case**, **const**, **div**, **do**, **downto**, **else**, **end**, **file**, **for**, **forward**, **function**, **goto**, **if**, **in**, **label**, **mod**, **nil**, **not**, **of**, **or**,

packed, **procedure**, **program**, **record**, **repeat**, **set**, **then**, **to**, **type**, **until**, **var**, **while**, **with**.

Además, en este libro se usarán las siguientes, añadidas en los compiladores de Turbo Pascal:

implementation, **interface**, **string**, **unit**, **uses**

Cada palabra reservada tiene un cometido específico que es inalterable; dicho de otro modo, las palabras reservadas no son redefinibles.

Identificadores

Los *identificadores* desempeñan un papel similar al de los sustantivos (representando objetos), adjetivos (representando tipos, que califican los objetos) y verbos (representando acciones) en las oraciones.

Los identificadores que están disponibles antes de empezar a escribir un programa se llaman *predefinidos*; damos la siguiente clasificación:

1. Archivos estándar de entrada/salida:

   `input`, `output`.

2. Constantes:

   `False`, `MaxInt`, `True`.

3. Tipos:

   `boolean`, `char`, `integer`, `real`, `text`.

4. Funciones:

   `Abs`, `ArcTan`, `Chr`, `Cos`, `EoF`, `EoLn`, `Exp`, `Ln`, `Odd`, `Ord`, `Pred`, `Round`, `Sin`, `Sqr`, `SqRt`, `Succ`, `Trunc`.

5. Procedimientos:

   `Dispose`, `Get`, `New`, `Pack`, `Page`, `Put`, `Read`, `ReadLn`, `Reset`, `Rewrite`, `Unpack`, `Write`, `WriteLn`.

La posibilidad de extensión del lenguaje permite la creación de identificadores (definidos por el programador) para representar archivos, constantes, variables, tipos, funciones y procedimientos a la medida de nuestras necesidades (véase la figura 4.1).

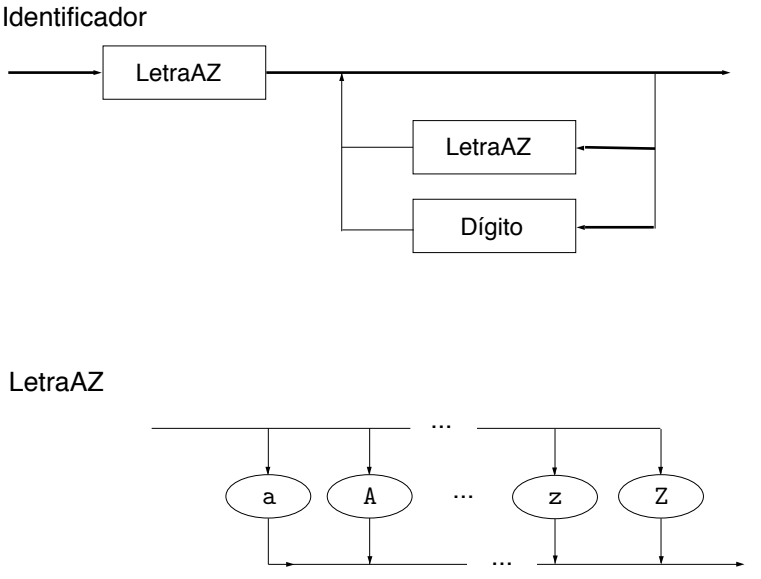


Figura 4.1.

Cualquier cadena de caracteres no resulta válida como identificador: existen razones para limitarlas. Los identificadores deberán estar formados por las letras[1] y los dígitos,[2] empezando por letra y sin espacios en blanco. En los identificadores, las letras mayúsculas y minúsculas son indistinguibles para el compilador.

Ejemplos de identificadores admitidos son:

`max`, `LimSup`, `anno`, `MaxCoorY`, `EsPrimo`, `EsValido`, `Seat600D`

En cambio, no son correctos:

`primero-F`, `Primero F`, `600D`

Por otra parte, se recomienda la elección de identificadores mnemotécnicos, que facilitan su uso al estar relacionados con el objeto que se nombra además de aumentar la legibilidad de nuestros programas. Por ello, no es conveniente que los identificadores sean excesivamente cortos (probablemente sin significado) ni excesivamente largos (lo que incomoda su escritura e interpretación, y puede

[1]En Pascal estándar (y también en Turbo Pascal) se excluyen la `ñ` y las vocales acentuadas.

[2]En Turbo Pascal también se permite el carácter de subrayado, luego son válidos los identificadores `Max_x` o `Area_circulo`. No obstante, se evitará su uso en este libro.

provocar errores). A propósito de esto, en el apartado 5.2 se amplían estas recomendaciones.

El lenguaje permite también dotar a los identificadores predefinidos con un nuevo significado; sin embargo, este cambio es en general poco recomendable, por lo que no consideraremos esta posibilidad.

Símbolos especiales

Son similares a los signos de puntuación de las oraciones:

```
+  -  *  /  :=  .   ,  ;  :   =  <  >  <=  >=  <>
(  )  [  ]  (*  *)  {  }  (.   .) ..
```

Los que están formados por varios caracteres deben escribirse sin espacios en blanco entre ellos.

Literales

En el curso de un programa, con frecuencia necesitamos escribir directamente elementos del dominio de un tipo básico para expresar cantidades numéricas enteras (como `365`) o reales (como `3.141592`), caracteres (como `'&'`) o cadenas de caracteres (como `'1 año = 365 días'`, en las que es posible intercalar espacios en blanco, así como el carácter `ñ`, las vocales acentuadas y otros símbolos, pero sin poderse partir una cadena entre más de una línea). Esos valores escritos directamente se llaman *literales*.

Entre los tipos básicos, los valores extremos del dominio de `integer` *no* se suelen expresar directamente con literales, sino mediante identificadores con un valor constante (por ejemplo `MaxInt`). Por otra parte, los literales que expresan cadenas de caracteres no pertenecen obviamente a ninguno de los tipos básicos, aunque existe la posibilidad de construir objetos apropiados (véanse los capítulos 11 y siguientes), y su uso es tan frecuente que algunas versiones concretas de Pascal proporcionan ese tipo ya predefinido (véase el apéndice B sobre Turbo Pascal).

Comentarios

Para facilitar la lectura de los programas, es una buena costumbre intercalar *comentarios* en castellano. Se distinguen del texto en Pascal por los delimitadores { y }, o bien `(*` y `*)`. Por ejemplo:

```
{Programa que halla la hipotenusa de un triángulo rectángulo}
  {Autor:  Pitágoras, presumiblemente}
  {Fecha:  hacia el s. VI a. C.}
```

Cuando se encuentra un símbolo { ó `(*`, todos los caracteres que lo siguen hasta el primer símbolo } ó `*)`, respectivamente, son ignorados, emparejándose { con } y `(*` con `*)`. El texto recorrido entre esos delimitadores es un *comentario*, y se interpreta como un espacio en blanco. Por consiguiente, no sería correcto interrumpir una palabra reservada, un identificador o un literal con un comentario.

Los comentarios en Pascal no pueden anidarse, es decir, no se puede poner un comentario dentro de otro.

4.2.1 Constantes y variables

En el ejemplo del apartado 4.1, se definió el valor de `Pi`: decimos que `Pi` es una *constante* para expresar que su valor no será alterado en el curso del programa; también se dice que es una constante *con nombre* para diferenciarla de las constantes expresadas literalmente, a las que también se llama constantes *anónimas*. En contraposición, los objetos `r` (radio) y `A` (área) pueden representar diferentes valores, por lo que se llaman *variables*.

En los objetos constantes y variables mencionados se pueden considerar los siguientes aspectos: el *identificador* (`Pi`, `r` y `A`), que es el término con el que pueden referirse; su *tipo* (`real`, en los tres casos) y el *valor* (`3.14` constantemente para `pi`, y desconocidos de antemano para `r` y `A`, variables en el transcurso del programa).

Para resaltar estos aspectos de los objetos constantes (con nombre) y variables, es corriente representarlos así:

entendiendo que el tipo determina el espacio en donde reside el valor, de forma que sólo son aceptables valores del correspondiente dominio.

4.3 Instrucciones básicas

4.3.1 Asignación

La *instrucción de asignación* se utiliza para dar un valor inicial a las variables o para modificar el que ya tienen.

En algunos compiladores, una variable declarada presenta un valor indefinido al iniciarse el programa; en efecto, se trata de un valor "basura", representado por el contenido de la memoria reservado cuando se declaró la variable. Lógicamente, un programa que depende de valores indefinidos tiene un comportamiento indeterminado; por ello es necesario evitar el operar con tales variables, asignándoles valores iniciales.

Una variable con valor indeterminado se puede representar así:

La asignación graba un valor en la memoria y destruye su valor previo, tanto si es un valor concreto como si es indeterminado. Consideremos la siguiente sentencia de asignación:

```
x1:= (-b + SqRt(Sqr(b) - 4 * a * c))/(2 * a)
```

Consta de un identificador de variable (`x1`), el símbolo de la asignación (que es `:=`) y una expresión. El proceso de asignación se produce de la siguiente forma: en primer lugar se evalúa la expresión, calculándose el valor final, y a continuación se almacena el valor en la memoria.

Si asignamos un valor (`1.74`, por ejemplo) a la variable, ésta pasaría a representarse así:

Ejemplos de instrucciones de asignación:

```
base:= 10.0
altura:= 20.0
area:= base * altura / 2
contador:= contador + 1
acumulador:= acumulador + valor
```

La sintaxis de una instrucción de asignación viene dada por el diagrama de la figura 4.2. Subrayamos que, semánticamente, el identificador debe representar una variable, y que el resultado de la expresión debe ser del mismo tipo que la variable.[3]

[3]Salvo que la expresión sea `integer` y la variable `real`. (Véase lo comentado en el apartado 3.3.)

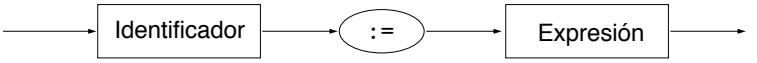


Figura 4.2. Instrucción de asignación.

Obsérvese la gran diferencia que existe entre una asignación (que es una acción y tiene el efecto de alterar el valor de una variable) y una igualdad, que es una proposición que afirma una relación entre objetos. Por ejemplo, la asignación

```
contador:= contador + 1
```

es una *instrucción* que tiene por objeto incrementar en una unidad el valor de la variable `contador`, mientras que la igualdad

```
contador = contador + 1
```

es una *expresión booleana* de una relación que, por cierto, es falsa cualquiera que sea el valor de `contador`.

4.3.2 Instrucciones de escritura

En todo programa se tienen entradas y salidas de datos con las que el programa se comunica con el exterior. La lectura o entrada de datos se realiza a través de dispositivos tales como el teclado, una unidad de disco, o fichas perforadas en los computadores antiguos, etc. La escritura o salida de resultados se realiza a través de dispositivos como la pantalla o la impresora.

Es habitual asumir un medio de entrada y uno de salida implícitos, que se utilizan mientras no se indique otro distinto. Frecuentemente en los computadores personales se adopta la consola como medio estándar, de manera que los datos se introducen a través del teclado y los resultados se escriben en el monitor.

La *salida* de resultados se expresa en Pascal con las órdenes `Write` y `WriteLn`, que pueden tener varios argumentos consistentes en expresiones de diferentes tipos:

```
Write(1 + 2 + 3)
WriteLn('Un tigre, dos tigres, tres tigres, ...')
WriteLn(1234, 56, 7)
WriteLn('El doble de ', n ,' es ', 2 * n)
```

El efecto de ambas se lleva a cabo en dos pasos sucesivos para cada uno de sus argumentos: en primer lugar se evalúa la expresión; en segundo se escribe el

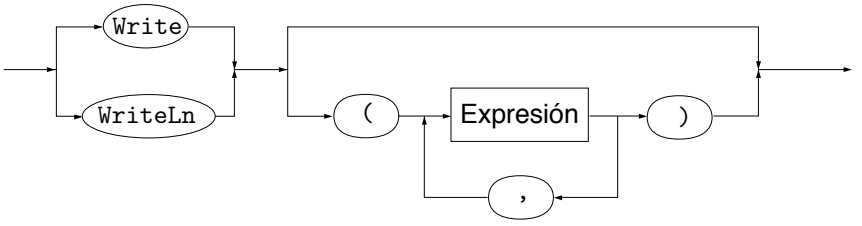


Figura 4.3. Instrucción de escritura.

resultado en el dispositivo de salida estándar. Los resultados de sus expresiones se escriben sin espacio de separación, a no ser que se dé explícitamente.

Estas instrucciones se diferencian en que la orden `WriteLn` genera un salto de línea, situando el cursor en el principio de la línea siguiente, listo para seguir la siguiente instrucción de escritura. Por ejemplo, si se efectuaran las cuatro instrucciones del ejemplo consecutivamente, la salida sería así:

```
6Un tigre, dos tigres, tres tigres, ...
1234567
El doble de 15 es 30
```

suponiendo que `n` es el entero `15`. El cursor salta y se queda en la cuarta línea, listo para continuar la escritura.

Ambas instrucciones pueden utilizarse sin argumentos: la instrucción `Write` no produce efecto alguno, mientras que `WriteLn` provoca un salto de línea. Por lo tanto, la secuencia de instrucciones

```
Write; Write('Hola'); WriteLn
```

equivale a la instrucción

```
WriteLn('Hola')
```

La sintaxis de estas instrucciones se describe en la figura 4.3.

Parámetros de formato de salida

- Con datos de tipo `integer`:

  La salida de resultados mediante `Write` y `WriteLn` está bastante limitada: incluso mediante el espaciado, los números quedan desalineados. Para resolver este problema se utilizan las salidas con formato añadiendo un

número entero a cada una de las expresiones por escribir, que indica al procedimiento `Write` o `WriteLn` en qué espacio debe justificar (por la derecha) cada uno de los valores numéricos. Por ejemplo, las instrucciones siguientes sitúan sus resultados correctamente sangrados:[4]

```
WriteLn(1234:5,56:5,7:5) ‖ ␣1234␣␣␣56␣␣␣␣7
WriteLn(12:5,345:5,67:5) ‖ ␣␣␣12␣␣345␣␣␣67
```

La salida puede rebasar el espacio reservado:

```
WriteLn(12345:3) ‖ 12345
```

- Con datos reales:

  Mientras no se indique lo contrario, la salida de valores reales se escribe en notación científica, que es bastante ilegible. Por ejemplo:

  ```
  2.7315190000E+02
  ```

  Como primera mejora de esta presentación, podemos justificar el resultado a la derecha, como se ha hecho con los datos `integer`:

  ```
  Write(a:15) ‖ ␣2.73151900E+02
  ```

  añadiendo a la izquierda los espacios en blanco necesarios.

  Aún mejor es añadir un doble formato, mostrándose el real en notación decimal: el primer parámetro indica las posiciones totales, como se ha visto, y el segundo el número de decimales

  ```
  Write(a:10:3) ‖ ␣␣␣273.152
  ```

  redondeando las cifras visibles si es preciso.

- Con caracteres y cadenas de caracteres:

  Los valores de tipo carácter pueden justificarse mediante un parámetro de formato que expresa el espacio mínimo total, justificando la salida a la derecha:

  ```
  WriteLn('A':8)      ‖ ␣␣␣␣␣␣␣A
  WriteLn('AEIOU':8)  ‖ ␣␣␣AEIOU
  ```

[4] Indicamos de este modo las instrucciones (a la izquierda) junto con la salida que producen en el `output` (derecha). Usaremos el símbolo ␣ para precisar el espacio ocupado por el carácter blanco.

- Con datos de tipo `boolean`:

  Se puede añadir un parámetro de salida a las expresiones booleanas que justifica el resultado por la derecha:

  ```
  WriteLn(ok:5) ‖ ␣TRUE
  ```

  donde se ha supuesto que el valor de `ok` es `True`.

El diagrama sintáctico de la figura 4.3 se puede completar trivialmente para que admita la posibilidad de incluir parámetros de formato.

El archivo output

Los resultados de un programa se escriben en el `output`, que frecuentemente es el monitor. En realidad, el archivo `output` consiste en una secuencia de caracteres (véase el apartado 14.3), por lo que los resultados numéricos se convierten en los caracteres que representan el correspondiente valor.

Entre esos caracteres existe una marca especial que representa el salto de línea (que suele representarse mediante ↵), así como otra para indicar el final del archivo (que representaremos mediante •). Por ejemplo, el final del `output` de los ejemplos anteriores puede representarse así:

```
...␣␣␣␣A↵␣␣␣AEIOU↵␣TRUE↵...•
```

En los dispositivos usuales el carácter ↵ se interpreta como un retorno de carro y un avance de línea, confiriéndole a la salida el aspecto global de una sucesión de líneas, de interpretación visual mucho más fácil que una sucesión de caracteres sin más.

4.3.3 Instrucciones de lectura

Las operaciones de entrada se realizan en Pascal mediante los procedimientos `Read` y `ReadLn`, cuya sintaxis se muestra en la figura 4.4. Como ejemplo de estas instrucciones tenemos:

```
Read(x,y,z)
ReadLn(u)
```

que actúan sobre una o más variables, estando separadas por comas cuando se trata de más de una.

Al llegar a esta instrucción, el computador lee los valores introducidos y los asigna por orden a las variables argumento indicadas. Debe señalarse que cada valor leído debe tener un tipo compatible con el de la variable en la que se almacena.

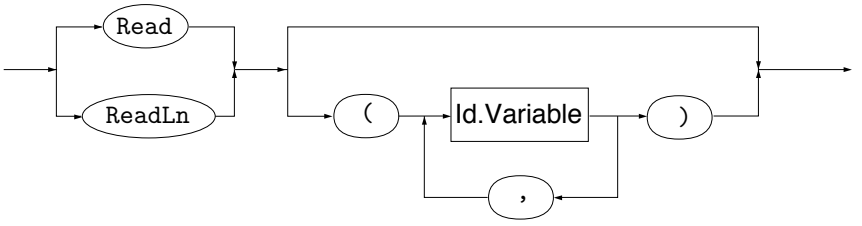


Figura 4.4. Instrucción de lectura.

El archivo input

Los datos del programa se leen del `input`, que frecuentemente es el teclado.

Surgen ahora tres cuestiones que deben aclararse. En primer lugar, el archivo `input` también consiste en realidad en una secuencia de líneas de caracteres (véase el apartado 14.3), que deben convertirse en números cuando las variables correspondientes sean de tipo numérico. Cuando haya varias variables numéricas, se pueden escribir en una sola línea separadas con espacios en blanco. En el ejemplo[5]

```
ReadLn(var1, var2, var3) ‖ 123␣456␣789↵
```

se asignaría `123` a `var1`, `456` a `var2` y `789` a `var3`, manteniendo el orden de lectura.

Por lo tanto, la introducción de datos, sean del tipo que sean, se realiza a través de una secuencia de caracteres. Es posible que esta conversión no se pueda llevar a cabo por incompatibilidad de tipos, produciéndose un error de ejecución. Así ocurriría en el ejemplo

```
Read(var1) ‖ a↵
```

si la variable `var1` fuera de tipo numérico.

En segundo lugar, el efecto de la sentencia `ReadLn` consiste en captar los datos del `input` y avanzar hasta rebasar el siguiente salto de fin de línea. Así por ejemplo, siendo las variables `a`, `b` y `c` de tipo numérico, en la siguiente situación

```
ReadLn(a, b);  ‖ 1␣2␣3␣4␣5↵
Read(c)        ‖ 6␣7␣8↵
```

[5]Ahora, la parte de la derecha representa el `input`.

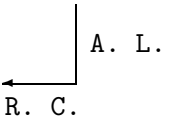
las variables `a`, `b` y `c` tomarían los valores 1, 2 y 6, respectivamente. Cuando se leen variables numéricas se saltan los blancos anteriores, que así actúan como separadores; en cambio, los caracteres se leen de uno en uno, sin separación de ningún tipo.

Subrayamos que, en realidad, el `input` consiste en una única tira de caracteres:

$$1\ 2\ 3\ 4\ 5\hookleftarrow 6\sqcup 7\sqcup 8\hookleftarrow \ldots \bullet$$

Finalmente, usaremos en adelante el símbolo • para expresar el fin del archivo de entrada de datos.

Conviene indicar que, cuando se trabaja en el sistema operativo DOS, el símbolo $\hookleftarrow$ representa a la vez el avance de línea (A. L.) y el retorno de carro (R. C.), tal como refleja su símbolo usual:



Sin embargo, en algunos traductores de Pascal para computadores personales, a veces se le atribuye el papel adicional de la cesión del control al computador (desde el teclado) para que reanude su actividad (véase el apéndice C). Esta coincidencia de papeles dificulta a veces observarlos aisladamente cuando el `input` es el teclado.

4.4 Partes de un programa

En este apartado se reúnen las ideas expuestas en los anteriores, presentándose las partes o secciones componentes de los programas en Pascal: encabezamiento, declaraciones y bloque o cuerpo de acciones.

Al fin disponemos de todos los elementos necesarios para escribir los primeros programas, aunque se trata de programas muy simples, carentes por supuesto de muchos mecanismos del lenguaje.

Por otra parte, además de reunir las ideas introducidas hasta ahora, el estudiante debería en este punto conseguir poner a punto sus primeros programas en un entorno de programación real.

4.4.1 Encabezamiento

El encabezamiento de un programa establece una identificación del mismo. En cierto modo equivale al título de un libro e incluye información sobre los

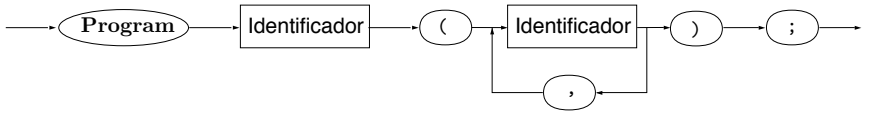


Figura 4.5. Encabezamiento de un programa.

objetos, externos al programa, con que éste intercambia información: la inclusión de éstos en el encabezamiento establece la comunicación correspondiente desde el programa.

En los primeros programas estos objetos son sólo los archivos estándar: el de entrada de datos (`input`) y el de salida (`output`), que en los computadores personales representan a la consola (teclado y monitor respectivamente); ambos archivos se incluirán siempre que el programa deba realizar operaciones de entrada y salida respectivamente, aunque conviene que el archivo `output` esté siempre presente, para indicar al computador dónde comunicar las eventuales situaciones de error. Más adelante, se verá cómo el programa podrá recibir información de otra procedencia (por ejemplo, una tabla estadística situada en un archivo de disco), o dirigir su salida a otros dispositivos (tales como la impresora).

El encabezamiento es obligatorio en Pascal estándar pero optativo en Turbo Pascal y en otros traductores; sin embargo, es recomendable utilizarlo siempre, para que los programas sean más claros y se puedan usar en otros entornos.

El encabezamiento empieza con la palabra reservada **Program**, seguida del nombre del programa, que debe ser un identificador válido de Pascal y, entre paréntesis, la lista de parámetros del programa. El encabezamiento se separa de las siguientes secciones con un punto y coma (`;`). Por ejemplo:

```
Program AreaCirculo (input, output);
Program DeclaracRenta (input, output, tablaRetenciones);
```

Así pues, la sintaxis del encabezamiento responde al diagrama de la figura 4.5.

4.4.2 Declaraciones y definiciones

Además de los identificadores predefinidos, el usuario casi siempre va a necesitar el uso de otros nuevos, en cuyo caso debe introducirlos y describirlos (excepto el identificador del programa) antes de usarlos; este protocolo se corresponde conceptualmente con la sentencia siguiente, de uso común en Matemáticas

$$\text{Sean } n \in \mathbb{Z}, x \in \mathbb{R}, p = 3.14 \text{ y } f : \mathbb{R} \to \mathbb{R} \text{ tal que } f(x) = px^n$$

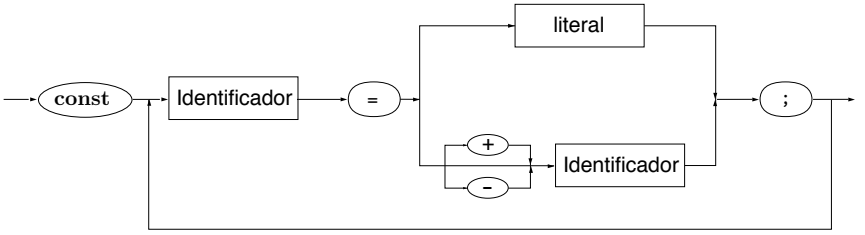


Figura 4.6. Definición de constantes.

legitimando el posterior uso de los identificadores n, x, p y f.

En Pascal, esta información de partida permite al compilador hacer las correspondientes asignaciones de memoria y verificar que el uso de todos esos objetos se ajusta a sus características, avisando al programador en caso contrario.

La obligación de incluir declaraciones sólo se da cuando el programador necesite incluir objetos nuevos para usarlos en el programa. Estos objetos responden a varias categorías: etiquetas, constantes, tipos, variables, procedimientos y funciones. Veremos con detalle cada una de ellas en su momento; por ahora, basta con introducir la definición de constantes y la declaración de variables.

Definición de constantes

El diagrama sintáctico de la definición de constantes aparece en la figura 4.6. En el apartado 4.2.1 se explicó la posibilidad de usar constantes literalmente, escribiendo su valor; a menudo interesa expresar constantes mediante identificadores (v.g. `Pi`). Para utilizar constantes con nombre, hay que definirlas previamente usando la palabra reservada **const**, del modo siguiente:

```
const
  TempCongelaAgua = 0;
  TempHierveAgua = 100;
  Pi = 3.14;
  MenosPi = - Pi;
  PrimeraLetra = 'A';
  Vocales = 'aeiou';
```

Una vez definidas las constantes, pueden intervenir en expresiones al igual que los literales. Por ejemplo: `Pi * Sqr(7.5)`.

En general, siempre que en un programa existan valores constantes, se recomienda su presentación mediante constantes con nombre. De este modo, resulta el programa más legible (v.g. `Pi`, `E`, `AnchoPantalla`, `AltoPantalla`).

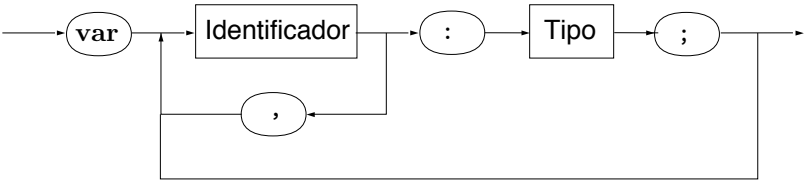


Figura 4.7. Declaración de variables.

Además, cuando una constante se repita en varios puntos de un programa, bastará con escribir una sola vez su valor, siendo así muy fácil modificar el programa adaptándolo a nuevos valores de las constantes.

Declaración de variables

La declaración de variables sigue el diagrama sintáctico de la figura 4.7, donde la palabra Tipo es (por el momento) un identificador de entre `integer`, `real`, `char` y `boolean`.

Como ya se ha dicho, las variables son objetos cuyo valor no se conoce *a priori*, o bien puede cambiar a lo largo del programa. En cambio, el tipo de las variables permanece inalterable desde que se establece al principio del programa; los traductores de Pascal utilizan esta información para determinar la cantidad de espacio reservado en la memoria para cada objeto y la forma en que se hará la representación, así como realizar las verificaciones aludidas en el apartado 3.7.

Para poder utilizar una variable es preciso declararla:

```
var
  indice, contador, edad: integer;
  altura, peso: real;
  esPrimo, hayDatos: boolean;
  inicial: char;
```

Deben recordarse las recomendaciones para elegir identificadores mnemotécnicos que guarden relación con el objeto al que dan nombre y que no sean excesivamente largos ni cortos.

4.4.3 Cuerpo del programa

En el cuerpo del programa es donde se relacionan las sucesivas sentencias o instrucciones ejecutables que componen el programa. Va precedido por la palabra reservada **begin** y termina con la palabra reservada **end** y un punto final, y las instrucciones se separan con puntos y comas (véase la figura 4.8).

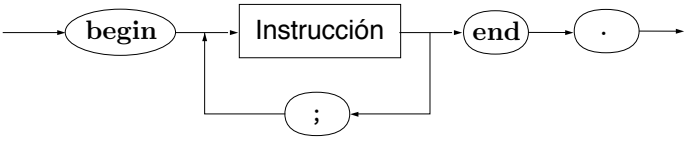


Figura 4.8. Cuerpo de un programa.

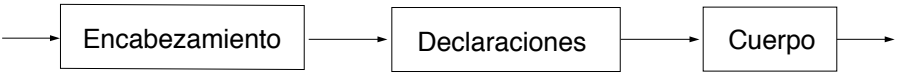


Figura 4.9. Estructura básica de un programa

Los principales tipos de sentencias ejecutables son los siguientes: instrucciones de asignación, de entrada o salida, estructuradas y llamadas a procedimientos. Estas dos últimas las estudiaremos en capítulos posteriores.

4.4.4 Conclusión: estructura general de un programa

Sintetizando lo dicho hasta ahora, un programa tiene tres partes: encabezamiento, declaraciones y cuerpo, según la figura 4.9

1. El encabezamiento del programa se considerará obligatorio.
2. Téngase en cuenta que la sección de declaraciones, en realidad, sólo tiene componentes optativas, por lo que puede haber programas sin declaración alguna.
3. Finalmente, el cuerpo o bloque es también obligatorio.

4.5 Ejercicios

1. Considerando el primer programa del capítulo,
   (a) Señale en él algunas palabras reservadas, símbolos especiales, identificadores predefinidos y definidos por el programador, literales y comentarios.
   (b) Localice las constantes que aparecen, ya sea con nombre o literales.
   (c) Delimite asimismo las tres secciones: encabezamiento, declaraciones y cuerpo.
2. Considérese el programa inicial de este capítulo. En el cuerpo de instrucciones, ¿es posible cambiar entre sí los identificadores `Pi` y `r`?

3. De las siguientes instrucciones de asignación, detecte las erróneas; averigüe el efecto de realizar las correctas y dé el tipo que deben tener las variables que intervienen en ellas para que sean aceptables.

(a)	`a:= 5`	(b)	`l:= 2 < 1`
(c)	`1:= a`	(d)	`p:= Sqr(2) = Sqr(2.0)`
(e)	`b:= 5 * a`	(f)	`MaxInt:= 32767`
(g)	`a:= a + 1`	(h)	`c:= 2 <= 7`
(i)	`p:= 2 * r`	(j)	`c1:= c2`
(k)	`'x':= 'y'`	(l)	`p:= q = r = s`

4. ¿Cuál es el efecto de llevar a cabo la siguiente lista de instrucciones sucesivamente?

$$(a)\texttt{a} := 2 \qquad (b)\texttt{b} := 3 \qquad (c)\texttt{a} := \texttt{b} \qquad (d)\texttt{b} := \texttt{a}$$

5. Efectúe lo siguiente, paso a paso, con varios pares `a` y `b` de tipo `integer`.

(a) `a:= a + b` (b) `b:= a - b` (c) `a:= a - b`

¿qué conclusión podemos extraer?

6. Considérese la siguiente fórmula (debida a Herón de Alejandría), que expresa el valor de la superficie S de un triángulo cualquiera en función de sus lados, a, b y c:

$$S = \sqrt{\frac{a+b+c}{2}\left(\frac{a+b+c}{2}-a\right)\left(\frac{a+b+c}{2}-b\right)\left(\frac{a+b+c}{2}-c\right)}$$

Dé una secuencia de instrucciones para obtenerla, evitando el cálculo repetido del semiperímetro, $sp = \frac{a+b+c}{2}$ y almacenando el resultado finalmente en la variable `S`.

7. Escriba una instrucción tal que, siendo `i` una variable `integer` con valor 10 y `x` una variable `real` con valor $1'234567 * 10^3$, ofrezca la salida

`v(10)␣=␣1234.57`

haciendo referencia a `i` y `x`.

8. Sean `a, b, c, d, e` y `f` variables reales. Escriba instrucciones de escritura que ofrezcan los valores de las siguientes variables y su suma:

$$\begin{array}{cccc} & a & c & e \\ + & b & d & f \\ \hline & a+b & c+d & e+f \end{array}$$

donde los valores de `a` y `b` están comprendidos entre 0 y 105, y su columna debe mostrarse redondeada, sin decimales; `c` y `d` están comprendidos entre 0 y 3, y su columna debe mostrarse redondeada con 2 decimales; `e` y `f` están comprendidos entre 0 y 3, y su columna debe mostrarse con un dígito, despreciando la parte decimal.

Obsérvese que el resultado ofrecido puede parecer erróneo, debido al redondeo o al truncamiento. Dé un ejemplo de ello.

9. Sean las instrucciones de lectura:

```
ReadLn(i1, c1, r1)
ReadLn(i2)
```

donde `i1, i2` son variables `integer`, `c1` es de tipo `char` y `r1` es `real`. Dados los `input`s

(a) `1A1`↵ `2␣3`↵ ...•

(b) `1␣A1`↵ `2␣`↵ ...•

(c) `1.2␣A1.2`↵ `1.2`↵ ...•

(d) `-1123␣␣0.5␣8`↵ `13`↵ ...•

diga cuáles son correctos y en su caso cuáles son los valores recibidos por las distintas variables. Explique por qué se producen los errores y la forma de subsanarlos.

10. Sean las variables `m` y `n` de tipo `integer`.

(a) Analice la compatibilidad de tipos en la siguiente expresión:

```
m < n or m = n
```

(b) Coloque los paréntesis necesarios para que sea correcta.

(c) ¿Puede ser correcta si `m` y `n` son de algún otro tipo?

Capítulo 5

Primeros programas completos

Con el conocimiento que ya se tiene de las instrucciones básicas de Pascal es posible desarrollar algunos programas completos que servirán para resaltar aspectos importantes que hay que tener en cuenta desde que se empieza a programar.

A continuación se presentan algunos programas sencillos, con los que se podrán efectuar las primeras prácticas, depurar errores y poner a punto los primeros programas, integrando las distintas fases que intervienen en la resolución de problemas mediante programas. Los problemas presentados como ejemplo y propuestos como ejercicios nos permitirán descubrir la necesidad de desarrollar los programas con naturalidad, afrontando las dificultades que planteen una a una y con una metodología que nos permita garantizar que el programa desarrollado es correcto.

A pesar de la simplicidad conceptual de los problemas resueltos en este capítulo, no debe desdeñarse una lectura sosegada del mismo para adquirir bien la base que se necesita para los capítulos posteriores.

5.1 Algunos programas sencillos

5.1.1 Dibujo de la letra "C"

El siguiente ejemplo es de lo más simple, aunque su efecto no produce tres líneas de asteriscos, como parecen indicar los identificadores definidos, sino una sola. Por la misma razón, los identificadores son inapropiados, ya que inducen a pensar que el programa escribe tres líneas distintas.

```
Program LetraC (output);
  {Dibujo de la letra 'C'}
  const  {Definición de constantes, que puede mejorarse}
    linea1 = '***';
    linea2 = '*';
    linea3 = '***';
begin {Cuerpo del programa}
  Write(linea1);
  Write(linea2);
  Write(linea3)
end.   {LetraC}
```

Es fácil modificarlo para que produzca efectivamente tres líneas (en vez de una sola) con 3, 1 y 3 asteriscos respectivamente, sustituyendo las instrucciones `Write` por `WriteLn`. Una mejora trivial consiste en dejarlo con sólo dos constantes, evitando repetirlas. Más aún, un programa tan sencillo no requiere definir esas constantes, pudiendo usarse directamente los literales.

Disposición clara de los programas. El programa anterior podría haberse escrito igualmente como sigue:

```
Program LetraC (output); {Dibujo de la letra 'C'}
const
{Definición de constantes, que puede mejorarse}
Linea1 = '***'; Linea2 = '*'; Linea3 = '***'; begin
{Cuerpo del programa} Write(Linea1); Write(Linea2);
Write(Linea3) end.
```

Sin embargo, la presentación inicial tiene ventajas indiscutibles: el programa inicial está dispuesto con claridad, con lo que leer, revisar y analizar el programa resulta más fácil. Asimismo, permite distinguir las componentes del programa y su estructura.

Como se ve, Pascal es muy flexible en cuanto a la disposición del texto de los programas, por lo que decimos que es un lenguaje *de formato libre*.

La misma finalidad tiene la colocación adecuada de los comentarios, que no tienen efecto alguno en el funcionamiento del programa y nos permiten en cambio incluir explicaciones sobre la finalidad del programa o sobre su funcionamiento.

En el mismo sentido, no está de más volver a recordar que se recomienda introducir identificadores mnemotécnicos, que sugieren el papel que juegan, facilitando también la lectura del programa.

El uso apropiado de una clara disposición del texto, la inclusión de comentarios apropiados, y una buena elección de los identificadores, se conoce como *autodocumentación*, y no es ningún lujo superfluo. Por el contrario, se considera preciso adquirir desde el principio el hábito de desarrollar programas claros y bien autodocumentados.

5.1.2 Suma de dos números

El siguiente programa halla la suma de dos números enteros:

```
Program Suma (input, output);
  {Pide dos enteros y halla su suma}
  var
    a, b: integer; {los sumandos}
begin
    {Lectura de los datos:}
  Write('Primer  número: ');
  ReadLn(a);
  Write('Segundo número: ');
  ReadLn(b);
    {Cálculos y resultados:}
  WriteLn(a,' + ',b,' = ',a + b)
end. {Suma}
```

Entradas y salidas claras. Además de la documentación interna del programa, otra recomendación que debe tenerse en cuenta desde el principio es que las lecturas de los datos y la salida de los resultados sean claras, incluyendo para ello los mensajes necesarios y ofreciendo comprobaciones de que los datos se han leído correctamente.

5.2 Programas claros ⇒ programas de calidad

Una recomendación de gran importancia para lograr que los programas sean correctos consiste en habituarse a escribirlos de forma clara, diferenciando bien sus distintos fragmentos para que sean fácilmente identificables y legibles. Otra

razón para que los programas sean claros es que un programa se escribe una vez, pero se lee muchas, bien para depurar sus posibles errores o para efectuar en él modificaciones. Se ha dicho que un programa bien escrito debe poderse leer tan fácilmente como una novela y, aunque esto puede resultar excesivo a veces, conviene esmerarse desde el principio en intentarlo antes incluso que perseguir la eficiencia.

En este sentido, conviene indicar que, de cara a mejorar la legibilidad de un programa, también es esencial una buena estructuración y organización de las acciones que lo componen, como se explica en los capítulos 6 y 7.

En la *documentación* de un programa se pueden observar, entre otros, los aspectos siguientes:

- El sangrado o encolumnado,[1] facilitando la identificación de fragmentos con distinto cometido o subordinados a otros, etc.

- Los comentarios, aclarando los siguientes detalles:

  - El cometido de los objetos introducidos.
  - El funcionamiento del programa.
  - Las condiciones requeridas o garantizadas en un determinado punto del programa. A este respecto, véase el apartado 5.4.

- La elección adecuada de identificadores, de forma que reflejen su contenido. Las siguientes indicaciones contribuyen a su rápida interpretación:

  - Como las constantes, variables y funciones representan objetos, suelen nombrarse con sustantivos (`Pi`, `x`, `sucesor`) o sintagmas nominales (`MaxInt` $\simeq$ máximo entero), excepto cuando representan valores lógicos, en que desempeñan el papel de sentencias (`esPrimo` $\simeq$ es primo), posiblemente abreviadas (`primo`, `Odd`, `ok`, `EoLn`).
  - Los procedimientos representan acciones, por lo que se les suele nombrar con verbos en infinitivo (`Escribir`, `Write`)

  Por otra parte, no debe escatimarse la longitud de los identificadores (aunque tampoco debe abusarse), cuando ello aclare el objeto identificado (`AreaTotal`, `PagaExtra`, `DistTierraSol`) incluso usando varias palabras para ello. En este caso, resulta aconsejable escribir con mayúscula la inicial de cada palabra. Esta recomendación es válida para los identificadores predefinidos (`WriteLn`, `SqRt`).

[1] En algunos manuales puede leerse "indentado", palabra que no existe en castellano.

También se suelen usar las mayúsculas y las minúsculas con un criterio uniforme, para que resulte sencillo interpretar la entidad de un identificador. Concretamente, la tipografía que seguimos para cada identificador es la siguiente:

- Constantes definidas, empezando con mayúscula: `Pi`, `N`, `Maximo`.
- Variables, empezando con minúscula: `x`, `miEdad`.
- Funciones y procedimientos, empezando con mayúscula: `SqRt`, `Write`.
- Tipos, empezando con minúscula. Los definidos por el programador, empezarán por `t` y luego seguirá una mayúscula.

Otra importante cualidad de los programas consiste en que las entradas de los datos y las salidas de resultados se efectúen también de forma clara, con mensajes concisos y apropiados, confirmando los datos capturados cuando su lectura sea delicada, haciendo uso de los parámetros de formato, etc.

5.3 Desarrollo descendente de programas

En este apartado desarrollaremos un programa que tiene por objeto hallar la hipotenusa de un triángulo rectángulo a partir de las longitudes de sus catetos. Procederemos en tres pasos:

1. *Obtención de los catetos.*
2. *Cálculo de la hipotenusa.*
3. *Escritura del resultado.*

Esta primera aproximación puede expresarse en un estilo muy próximo a Pascal:

```
Program Cálculo de hipotenusa
begin
  Obtener los catetos, catA , catB
  Hallar la hipotenusa, hipo
  Escribir el resultado, hipo
end.
```

Ahora, en una primera fase se desarrollan un poco estas acciones. Algunas son tan sencillas que pueden transcribirse directamente en Pascal, aunque pueden mantenerse los comentarios para indicar el cometido de cada segmento de programa:

```
Program Hipotenusa (input, output);
begin
    {Obtención de datos}
  Write('Catetos:  ');
  ReadLn(catA, catB);
  Hallar la hipotenusa, hipo
    {Escritura de resultados}
  WriteLn(' Hipotenusa = ', hipo)
end.   {Hipotenusa}
```

añadiendo entonces los identificadores que van surgiendo:

```
var
  catA, catB,          {catetos}
  hipo        : real; {hipotenusa}
```

Otras instrucciones en cambio son algo más complicadas, pudiendo descomponerse en varias más sencillas. Así por ejemplo, el paso *Hallar la hipotenusa* puede llevarse a cabo en dos:

Hallar la suma de los cuadrados de los catetos (SumCuadr)
Hallar la raíz de SumCuadr*, que es ya la hipotenusa*

que pueden escribirse directamente como las instrucciones siguientes:

```
sumCuadr:= Sqr(catA) + Sqr(catB)
hipo:= SqRt(sumCuadr)
```

requiriéndose añadir la variable sumCuadr, de tipo real.

Finalmente, este desarrollo desemboca en un programa en Pascal:

```
Program Hipotenusa (input, output);
  {Este programa pide las longitudes de los catetos de
  un triángulo rectángulo y halla la correspondiente hipotenusa}
  var
    catA, catB,          {longitudes de los catetos}
    sumCuadr,            {para guardar CatA^2 + CatB^2}
    hipo        : real;  {longitud de la hipotenusa}

begin {Prog. hipotenusa}
    {Obtención de los datos y su comprobación:}
  Write ('Introduce las longitudes de los catetos:  ');
  ReadLn (catA, catB);
  WriteLn ('un cateto mide ', catA:8:4, ' y el otro ', catB:8:4);
```

```
    {Cálculos:}
  sumCuadr:= Sqr(catA) + Sqr(catB);
  hipo:= SqRt(sumCuadr);
    {Resultados:}
  WriteLn ('La hipotenusa mide: ', hipo:8:4)
end.  {Prog. Hipotenusa}
```

Resumen

El programa `Hipotenusa` se ha desarrollado en fases sucesivas, a partir de un boceto a grandes rasgos del mismo. En esta primera versión aparecen acciones y datos escritos en los términos del propio problema.

Entonces empieza un proceso de refinamiento por pasos sucesivos, desarrollando esas acciones en cada fase: las más sencillas podrán escribirse directamente en Pascal; otras requerirán varios pasos en esa dirección. En ambos casos pueden aparecer nuevos objetos que será preciso incluir en las secciones de constantes o variables.

Este modo de proceder se llama *diseño descendente* y *refinamiento por pasos sucesivos*, ya que el desarrollo de un programa se lleva a cabo identificando primero grandes acciones y descendiendo a sus detalles progresivamente. En el apartado 7.3 se desarrollan estas ideas ampliamente.

5.4 Desarrollo de programas correctos

En este apartado se establece la base necesaria para razonar sobre la corrección de los programas durante su desarrollo, en vez de hacerlo *a posteriori*.

Por el momento, atendemos al efecto que tienen las instrucciones elementales sobre los datos manejados por el programa. De hecho, se puede garantizar que un programa es correcto cuando el efecto que produce a partir de unos datos genéricos consiste en desembocar en los resultados deseados. En suma, un programa es *correcto* si cumple con su cometido para unos datos cualesquiera, o sea, genéricos.

Nuestra propuesta consiste en habituarse, desde el principio, a concentrar la dosis de atención necesaria para estar seguro de que un programa es correcto. Para ello, se debe seguir este principio durante todas las fases del desarrollo.

5.4.1 Estado de los cómputos

El modelo de programación adoptado en este libro es el imperativo (véase el apartado 5.1.3 de [PAO94]). En él, el efecto concreto de una instrucción en

un momento dado puede variar dependiendo del conjunto de valores asociados a los objetos (constantes y variables) en ese instante. Esos valores (así como el conjunto de los datos de entrada y los resultados producidos) constituyen la noción de *estado* de un programa, que se altera mediante cualquiera de las sentencias básicas a medida que avanza la ejecución del programa.

Por ejemplo, si consideramos declaradas `a, b: integer`, el efecto de la instrucción `Write(a + b)` depende del estado (valor) de ambas variables, `a` y `b`, por lo que su interpretación en los puntos del programa donde aparezca

```
...
ReadLn(a, b);
WriteLn(a + b);
a:= a + b;
WriteLn(a + b);
b:= Sqr(a);
WriteLn(a + b);
...
```

requiere una interpretación histórica basada en los sucesivos estados precedentes. En concreto, si el `input` consiste en una línea con los números `2 3`↵, las instrucciones `WriteLn(a + b)` producen tres salidas distintas: `5`, `8` y `30`.

Trazado y depuración de un programa

Una forma de comprobar el comportamiento de un programa para un juego concreto de datos de entrada consiste en simular su funcionamiento a mano y seguir la evolución de los estados por los que atraviesa. Típicamente, esos estados incluyen información sobre el `input` por leer, el `output` emitido, los valores de las variables que intervienen y el punto en que estamos en un momento dado (véase el apartado 1.2.2).

Siguiendo con el fragmento de programa anterior, el estado de los cómputos se puede mantener en una tabla como la de la figura 5.1, en la que las posiciones representan los estados sucesivos entre las instrucciones.

Así, resulta fácil analizar cómo las instrucciones modifican el valor de las variables. Sin embargo, este sencillo método se vuelve inviable a poco que un programa se complique. Por ello, algunos entornos de desarrollo de programas incorporan facilidades para efectuar este seguimiento de forma automática, permitiendo establecer las variables o expresiones de nuestro interés, así como las posiciones en que deseamos que se produzca una suspensión momentánea de los cómputos para examinar su estado. Entre esos entornos se encuentra Turbo Pascal (véase el apartado C.2.6).

Posición	Input	Output	a	b
		...		
1	[2␣3↵]	[ ]	?	?
2	[ ]	[ ]	2	3
3	[ ]	[5↵]	2	3
4	[ ]	[5↵]	5	3
5	[ ]	[5↵ 8↵]	5	3
6	[ ]	[5↵ 8↵]	5	25
7	[ ]	[5↵ 8↵ 30↵]	5	25
		...		

Figura 5.1.

Precondiciones, postcondiciones y especificaciones

Otro inconveniente del seguimiento descrito es que sólo nos permite examinar el funcionamiento de un programa para un juego de datos concreto, de donde no podemos concluir que un programa es correcto para cualquier juego de datos de entrada. Para examinar la corrección de un programa, debemos caracterizar *en general* los puntos delicados, aportando una descripción (más o menos formal) del estado de los cómputos en ese punto.

En general, si tras la instrucción de lectura los valores de las variables x e y son x_0 e y_0 respectivamente, tenemos:

```
...;
```
$\{x = ?, y = ?\}$
```
Write('Números: ');
ReadLn(x, y);
```
$\{x = x_0, y = y_0\}$
```
x:= x + y;
```
$\{x = x_0 + y_0, y = y_0\}$
```
WriteLn(x,y);
```
$\{x = x_0 + y_0, y = y_0\}$
```
y:= Sqr(x);
```
$\{x = x_0 + y_0, y = (x_0 + y_0)^2\}$
```
WriteLn(x,y);
```
$\{x = x_0 + y_0, y = (x_0 + y_0)^2\}$
```
...
```

Los comentarios insertados ahora constituyen afirmaciones sobre el estado de los cálculos en un momento dado, y tienen una función doble:

- Nos permiten analizar con detalle el funcionamiento de un programa o un fragmento de programa. Por ejemplo, consideremos el programa siguiente, cuyo objeto consiste en intercambiar el valor de dos variables de tipo `char` entre sí:

```
Program Intercambio (input, output);
  var
    c1, c2: char; {los dos caracteres}
begin
  Write ('Caracteres:  ');
  ReadLn(c1, c2);
  c1:= c2;
  c2:= c1;
  WriteLn ('Invertidos: ', c1, c2)
end.   {Intercambio}
```

Si llamamos a y b a los caracteres leídos del `input`, se pueden insertar los siguientes predicados:

```
  {c1 = a, c2 = b}
c1:= c2;
  {c1 = b, c2 = b}
c2:= c1;
  {c1 = b, c2 = b}
```

con lo que *no* se obtiene el resultado deseado.

En cambio, el siguiente programa sí consigue llevar a cabo el intercambio de dos caracteres cualesquiera. La prueba de ello se da en el propio programa:

```
Program Intercambio (input, output);
  var
    c1, c2,          {los dos caracteres}
    aux     : char; {variable auxiliar}
begin
  Write ('Caracteres: ');
  ReadLn(c1, c2);
      {c1 = a, c2 = b}
  aux:= c1;
      {c1 = a, c2 = b, aux = a}
  c1:= c2;
      {c1 = b, c2 = b, aux = a}
  c2:= aux;
      {c1 = b, c2 = a, aux = a}
  WriteLn ('Invertidos: ', c1, c2)
end.   {Intercambio}
```

- En el razonamiento anterior, se ha partido de un programa y, para verificar su funcionamiento, se han incluido aserciones sobre el estado de los cálculos, averiguando así el efecto que tienen una o varias instrucciones sobre los mismos.

  Recíprocamente, se puede partir del efecto que un (fragmento de) programa debe producir para buscar instrucciones que lo logren. Por ejemplo, se puede plantear la búsqueda de un fragmento de programa I que modifique el valor de las variables `x, y, z: integer` del siguiente modo:

$$\begin{array}{l}\{x = x_0, y = y_0, z = z_0\}\\ I\\ \{x = y_0, y = z_0, z = x_0\}\end{array}$$

Las situaciones del estado inmediatamente anterior a la ejecución de una instrucción e inmediatamente posterior a la misma se llaman *precondición* y *postcondición* respectivamente.

El planteamiento anterior es la *especificación* formal de un problema en forma de ecuación, que puede leerse así: se pide un algoritmo I tal que, si se ejecuta cuando $x = x_0,\ y = y_0,\ z = z_0$, a su término se obtiene $x = y_0,\ y = z_0,\ z = x_0$. Para interpretar correctamente una especificación deben tenerse en cuenta además las declaraciones de los objetos involucrados.

En resumen, pueden entenderse las instrucciones como funciones que convierten un estado en otro. Las condiciones por las que atraviesa la ejecución de un programa pueden especificarse más o menos formalmente como comentarios, ayudándonos a comprender el funcionamiento de un algoritmo y a verificar su corrección.

Recíprocamente, es posible plantear un problema especificando las condiciones inicial y final que el algoritmo solución debe verificar. Este modo de proceder nos lleva a desarrollar algoritmos correctos, ya que el razonamiento sobre su funcionamiento no surge *a posteriori*, sino durante el proceso de desarrollo. Por supuesto, encontrar algoritmos que verifiquen una especificación dada requiere cierta experiencia en el desarrollo de los mismos; precisamente en ello consiste la programación.

Un medio aconsejable para adquirir el hábito de desarrollar algoritmos disciplinadamente consiste en esforzarse por razonar sobre la corrección de los programas desde el principio, como se ha indicado en el punto primero, incluyendo posteriormente la especificación de las condiciones necesarias en el desarrollo de los algoritmos.

5.4.2 Desarrollo descendente con especificaciones

Incorporando aserciones en el proceso descendente de programación, resulta que también las especificaciones van refinándose, indicando el cometido de cada parte del programa y las condiciones iniciales en que se ejecuta cada pieza del programa y las finales a su término. Así por ejemplo, el paso *"Obtener los catetos,* `catA` y `catB`", tiene por objeto cambiar el valor de las variables `catA` y `catB`, que es desconocido, y anotar en ellas los valores (digamos que a y b son los valores verdaderos dados en el `input`), lo que se expresa así:

$\{\texttt{CatA} = ?, \texttt{catB} = ?\}$
Obtener los catetos, `CatA` *y* `catB`
$\{\texttt{CatA} = a, \texttt{catB} = b\ \}$

De hecho, la frase *"Obtener los catetos,* `CatA` *y* `catB`*"* refleja precisamente el cometido de su especificación.

De esta forma, la primera fase del desarrollo del programa anterior puede escribirse así:

Program *Cálculo de hipotenusa*
begin
  {`CatA` =?, `catB` =?}
  Obtener los catetos, `CatA` *y* `catB`
  {`CatA` = a, `catB` = b }
  Hallar la hipotenusa, `Hipo`
  {`Hipo` = $\sqrt{a^2 + b^2}$ }
  Escribir el resultado, `Hipo`
  {`Output` = $\sqrt{a^2 + b^2}$ }
end.

En las fases sucesivas se irá refinando el algoritmo, que se va convirtiendo poco a poco en un programa.

5.5 Observaciones finales

Limitaciones del tipo integer. Se ha escrito y ejecutado el programa siguiente en un compilador en que `MaxInt` es `32767`. La ejecución del mismo está representada en la columna de la derecha:

```
Program LimitacionesDeInteger (output);
  {se asume la constante MaxInt = 32767}
  var
    n:  integer;
begin
  n:= 10000;
  WriteLn(n);    {10000}
  n:= n*4;
  WriteLn(n);    {-25536}
  n:= n div 4;
  WriteLn(n)     {-6384}
end. {LimitacionesDeInteger}
```

Se observa aquí que al ser $\mathcal{Z}$, el dominio de `integer`, distinto de $\mathbb{Z}$, las operaciones correspondientes también tienen sus limitaciones. En el ejemplo anterior, se produce un desbordamiento en la segunda instrucción del programa.

Limitaciones del tipo real. El siguiente programa tiene un resultado imprevisto: aunque la expresión `Ln(Exp(1)) = 1` es cierta, su evaluación produce el valor `False` en el siguiente programa:

```
Program LimitacionesDeReal (output);
begin
  ...
  WriteLn(Ln(Exp(1)) = 1)  { False}
  ...
end.   {LimitacionesDeInteger}
```

Ello se debe a que los números del tipo `real` se representan sólo aproximadamente: a título de ejemplo, la cantidad 0.1 es en binario un decimal periódico, por lo que se truncan cifras en su representación (véase el apartado 2.2.3 de [PAO94]). En esta referencia se identifican los peligros más frecuentes que surgen al trabajar con números reales, como es en concreto la comparación de cantidades reales.

Variables sin valor inicial. En el siguiente programa se hace uso de variables a las que no se ha dado valor alguno.

```
Program VariablesSinValorInicial (output);
  var
    x, y:  real;
begin
    {x =?, y =?}
  WriteLn(x);
  WriteLn(y);
  WriteLn(x+y)
end.  {VariablesSinValorInicial}
```

Por lo tanto, su funcionamiento produce resultados arbitrarios, como se muestra a continuación, donde cada columna representa una ejecución distinta.

3.4304031250E+04	2.0689620625E-36	0.0000000000E+00
9.2923334062E+18	1.6805384925E+24	8.5444437667E+37
2.6086154722E-04	-0.0000000000E+00	1.9225485289E-17

La conclusión es ésta: si no se da a las variables valor inicial, éstas toman valores arbitrarios.

5.6 Ejercicios

1. Escriba programas que den cada una de las siguientes salidas:

2. Enmiende los errores cometidos en la escritura del siguiente programa:

```
program Par-o-impar (imput); {Este programa
pide un entero:  si es par contesta "TRUE", y si
no, "FALSE" } beguin {datos} Write('Dame un
entero y averigüaré si es par); ReadLn(n);
{cálculos} esPar:= not odd n {resultados}
WriteLn('Solución:  '  esPar); end
```

3. Escriba un programa para cada uno de los siguientes problemas, documentándolo debidamente. Incluir además bajo el encabezamiento la información concerniente al autor (código, grupo, nombres, etc.), así como la fecha de realización. Una vez acabados, obténgase una copia escrita de ellos.

   (a) Solución de una ecuación de la forma $ax + b = 0$, supuesto a$\neq$0.

   (b) Lectura de los coeficientes a, b y c de una ecuación de segundo grado $ax^2 + bx + c = 0$ y cálculo de sus raíces, en los siguientes casos:

      i. Suponiendo que las raíces son reales. Ejecutarlo con los siguientes juegos de datos:

```
1   2    1
1   0   -1
1   0    0
```

      ii. Suponiendo que son imaginarias. Ejecutarlo con el siguiente juego de datos:

```
1   1   1
```

   (c) Desarrolle un programa que, para una cierta cantidad de dinero (en pesetas), da el cambio apropiado, en billetes de mil, monedas de quinientas, de cien, de veinticinco, de duro y de peseta.

$$cantidad \begin{cases} bill1000 \\ resto1000 \begin{cases} mon100 \\ resto100 \ \ldots \end{cases} \end{cases}$$

   (d) Convierta una cantidad de tiempo (en segundos, $\mathbb{Z}$), en la correspondiente en horas, minutos y segundos, con arreglo al siguiente formato:

```
3817 segundos = 1 horas, 3 minutos y 37 segundos
```

4. Escriba un programa que, en primer lugar, lea los coeficientes a_2, a_1 y a_0 de un polinomio de segundo grado

$$a_2x^2 + a_1x + a_0$$

y escriba ese polinomio. Y, en segundo, lea el valor de x y escriba qué valor toma el polinomio para esa x.

Para facilitar la salida, se supondrá que los coeficientes y x son enteros. Por ejemplo, si los coeficientes y x son 1, 2, 3 y 2, respectivamente, la salida puede ser

```
1x^2 + 2x + 3
p(2) = 9
```

5. Razone, informalmente, la corrección o falsedad de los siguientes fragmentos de programa, cuyo cometido se indica a continuación.

   (a) Intercambio del valor de dos variables enteras.

```
x:= x+y;
y:= x-y;
x:= x-y
```

   (b) Dado un entero n, hallar el menor $m \geq n$ que es par.

```
m:= n div 2 * 2
```

6. Los términos de la sucesión de Fibonacci[2] se definen así:

   - Los dos primeros son unos.
   - Cada término es igual a la suma de los dos anteriores

   Es decir: $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$

   Suponiendo que `a` y `b` son dos términos consecutivos de esa sucesión, razone la corrección de los siguientes fragmentos de programa desarrollados para hacer avanzar un paso esos términos:

   (a) Siendo `aux: integer` una variable auxiliar para hacer el trasvase de valores:

```
aux:= a; a:= b; b:= aux + a
```

   (b) Sin usar variable auxiliar alguna:

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

[2]Descubierta por Leonardo da Pisa (1180-1250) y publicada en su *Liber Abacci*, en 1202.

Tema II

Programación estructurada

Capítulo 6

Instrucciones estructuradas

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 sóla 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**

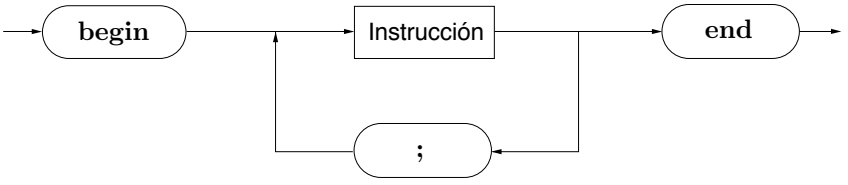


Figura 6.1. Diagrama sintáctico de una instrucción compuesta.

Téngase en cuenta que la interpretación del punto y coma es la de nexo 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).

```
begin
   Read(x);
   Read(y);
   begin
      x:= x + 1;
      y:= y + 2
   end;
   WriteLn(x * y)
end
```

```
begin
   Read(x);
   Read(y);
   x:= x + 1;
   y:= y + 2;
   WriteLn(x * y)
end
```

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;
```

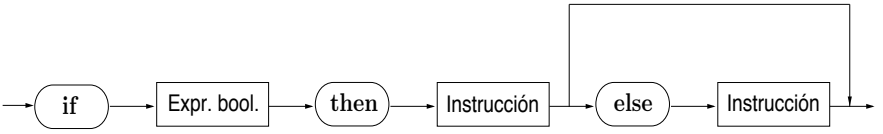


Figura 6.2. Diagrama sintáctico de **if-then-else**.

```
    {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

[1]Hay 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

if `C1` **then if** `C2` **then** `I2` **else** `I3` (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:

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

[2]El 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

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

o bien

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

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**.

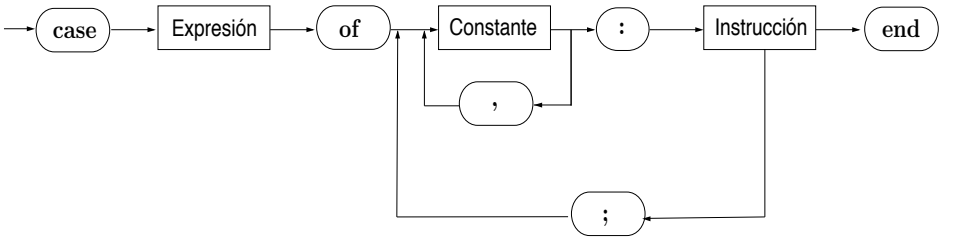


Figura 6.3. Diagrama sintáctico de la instrucción **case**.

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');
```

[3]Esto 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)
```

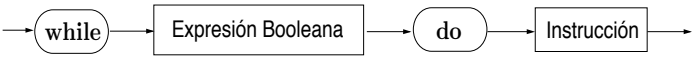


Figura 6.4. Diagrama sintáctico de la instrucción **while**.

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

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

[5] En 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.

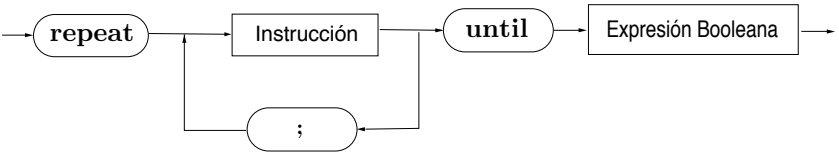


Figura 6.5. Diagrama sintáctico de la instrucción **repeat**.

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 *pre*probado de **while** hace que este consejo sólo sea válido para gente *pre*visora 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 >= 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 <= 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

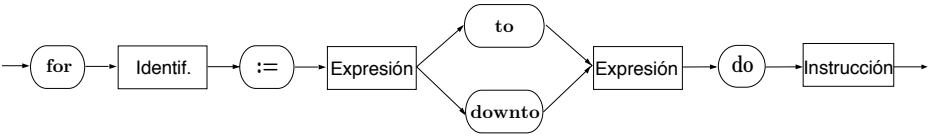


Figura 6.6. Diagrama de flujo de las instrucciones **for**.

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

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 $1, 2, \ldots, n$.

```
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 $0°$, $5°$, $\ldots$, $90°$.

```
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 $a_{ij} = \dfrac{i+j}{2}$:

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

[6]En 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 seudocódigo 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.

[7] La 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 precondición, 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 precondición 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 `ddo = dsor * coc + resto` verificando que `0` $\leq$ `resto` $\leq$ `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: `ddo = dsor * coc + resto`.

   (b) Se finaliza cuando `resto<dsor`. Al escribir el programa usaremos esta aserción para asegurar la corrección del código.

Ya podemos dar una primera aproximación al programa usando seudocódigo 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 <= 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 ≥ 0}
  while resto >= dsor do begin
    resto:= resto - dsor;
    coc:= coc + 1
  end; {while}
    {PostC.:  ddo = dsor * coc + resto y 0 <= 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

```
dsor * (coc + 1) + (resto - dsor)  =
 dsor * coc + dsor + resto - dsor  =
              dsor * coc + resto   =  ddo
```

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.

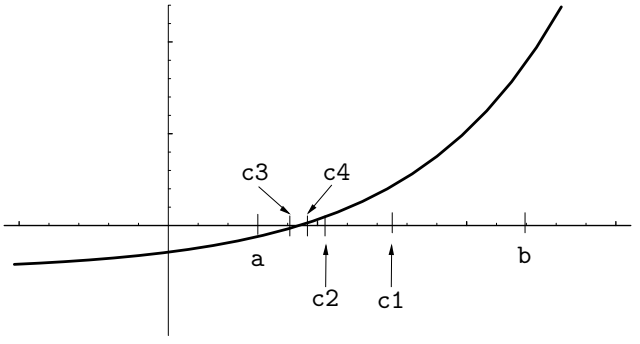


Figura 6.8. Aproximación por bipartición.

6.5 Dos métodos numéricos iterativos

Dada una función $f : \mathbb{R} \to \mathbb{R}$, 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 $\mid x - x_0 \mid < \varepsilon$, 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 * 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 seudocódigo 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 $f : \mathbb{R} \to \mathbb{R}$ 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.

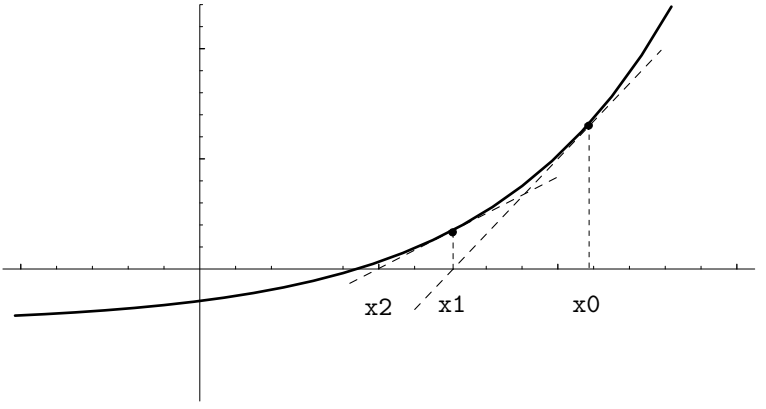


Figura 6.9. Aproximación por el método de Newton-Raphson.

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

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

[8] Pues 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

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

   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

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

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 $\{5, 16, 8, 4, 2, 1, 4, 2, 1, \ldots\}$ 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 $\dfrac{n!}{(n-k)!k!}$

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

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

11. **Integración definida.** Sea la función $f : \mathbb{R} \longrightarrow \mathbb{R}$. 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 $\int_0^{\pi} \text{sen}(x)dx$ 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:

$$(123\underline{4}, 5678) \rightarrow (123, 5678\underline{4})$$

    (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.

Ejemplo:	$6 = 1 + 2 + 3$
Contraejemplo:	$12 \neq 1 + 2 + 3 + 4 + 6$

(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 $\text{arcsen}(\frac{1}{5})$, 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

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.

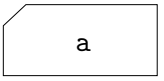
Para comprender mejor los problemas que surgen del uso incorrecto de los diagramas de flujo es necesario conocerlos un poco. Un diagrama de flujo se compone de bloques (que representan las acciones y las decisiones) y de líneas (que indican el encadenamiento entre los bloques).

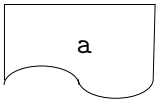
Los bloques de un diagrama de flujo pueden ser de cuatro clases distintas:

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

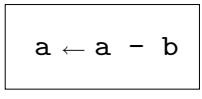
[1] El sentido de este adjetivo se explicará en el apartado 7.2.2.

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

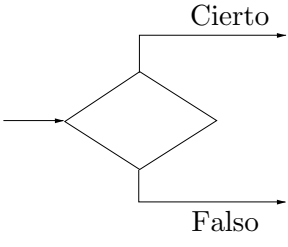


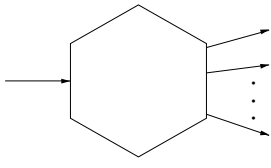


- Bloques de *procesamiento* de datos, que realizan operaciones con los datos leídos o con datos privados. Se representan mediante rectángulos que encierran la especificación del proceso, como por ejemplo:



- 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

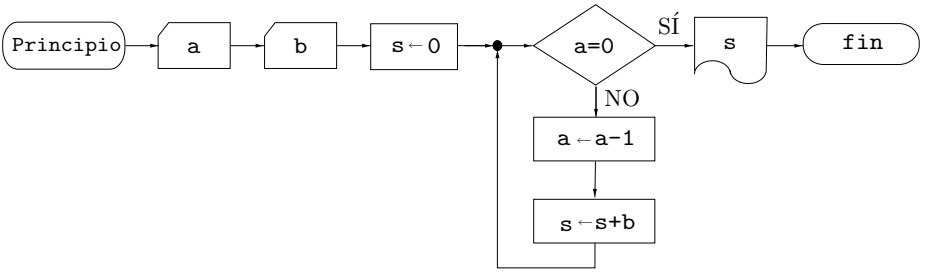


Figura 7.1. Diagrama de flujo para el producto de números naturales.

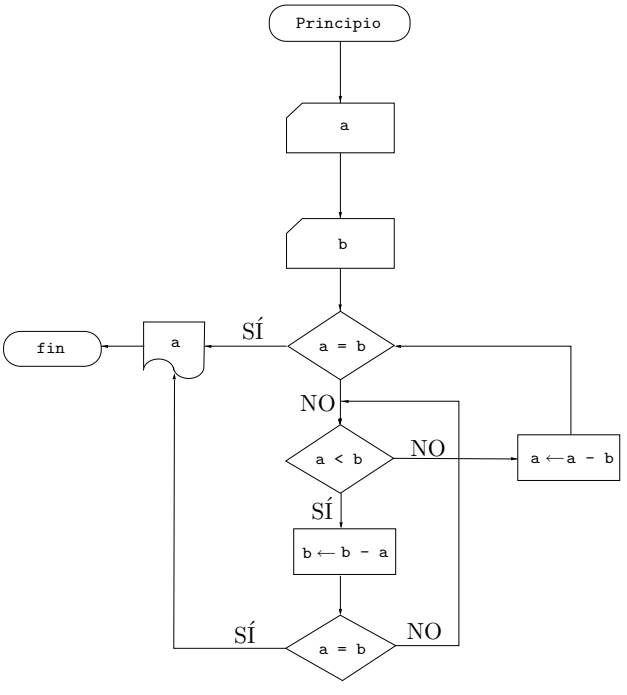


Figura 7.2. Un diagrama de flujo algo confuso.

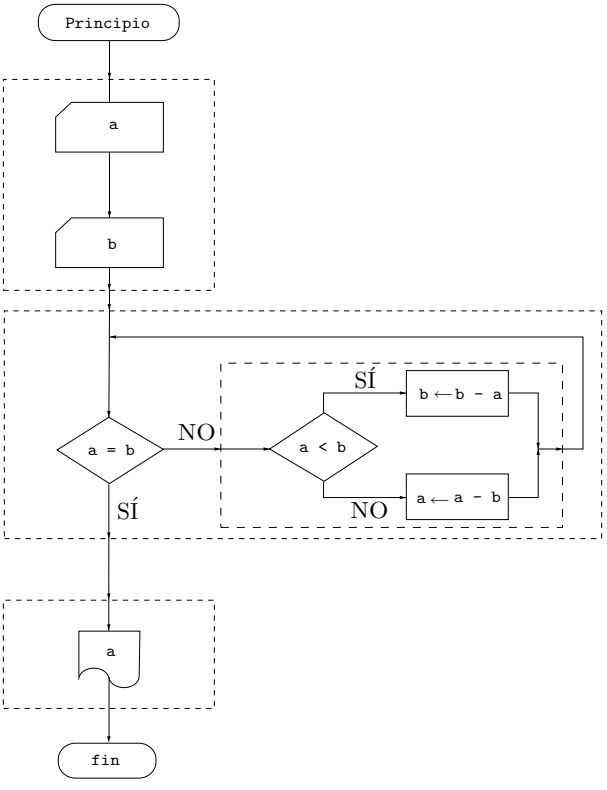
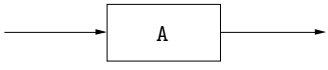


Figura 7.3. Versión bien organizada del diagrama anterior.

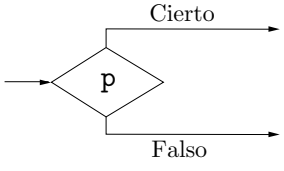
metodología será necesario disponer de una cierta clase de diagramas permitidos a partir de los cuales construir la teoría. Entre éstos se destaca la subclase de los diagramas *propios*, que representan, desde cierto punto de vista, a los programas correctamente estructurados.

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:

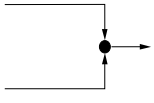
- *Acción*, que sirve para representar una instrucción (por ejemplo de lectura, escritura, asignación...).



- *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.

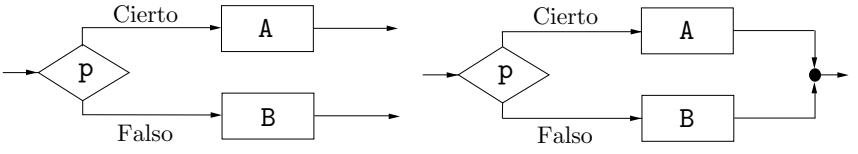


Figura 7.4. Un diagrama no propio y un diagrama propio.

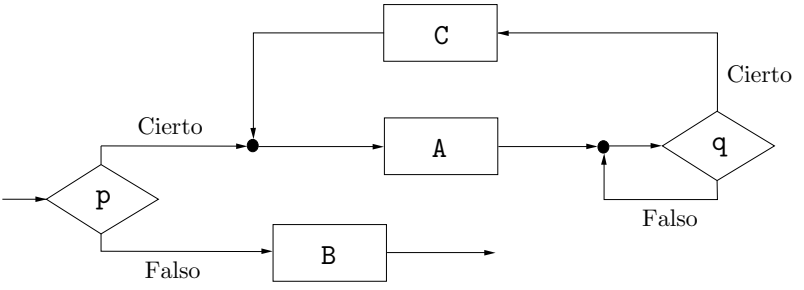


Figura 7.5. Un diagrama no propio.

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

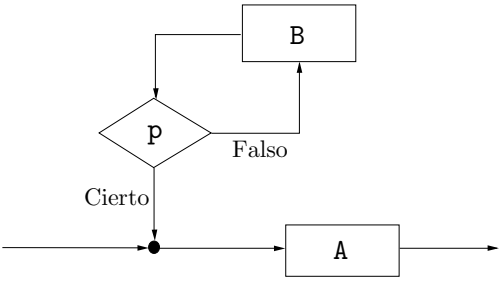
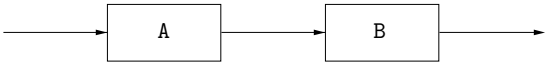


Figura 7.6. Un diagrama con elementos inaccesibles.

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:

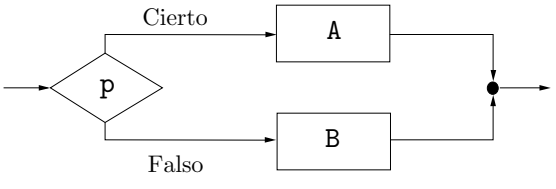


(7.1)

   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).

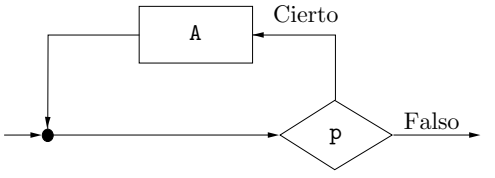


(7.2)

   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`.



(7.3)

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

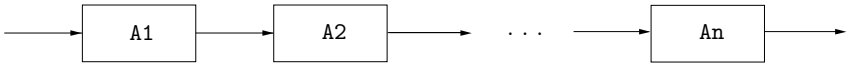
En esta construcción, el predicado `p` actúa como un control sobre la iteración, esto es, si se verifica `p` entonces se ejecuta `A`.

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**.

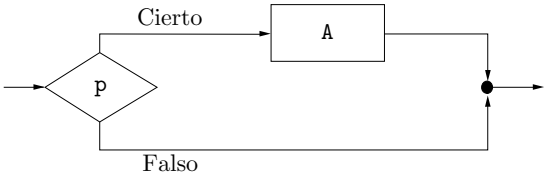
Otros diagramas de uso frecuente

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 `A1`, ..., `An`:



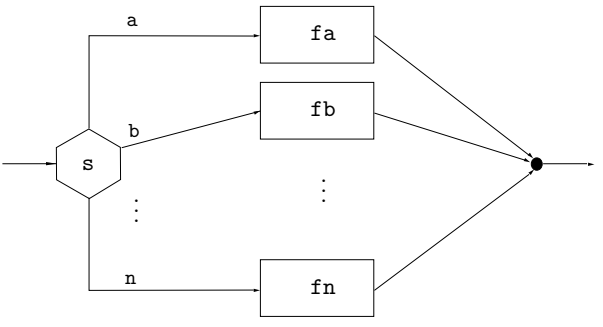
Si hacemos uso de la acción vacía (que se corresponde con una instrucción que no hace nada) se puede considerar la siguiente variante de la selección:



(7.4)

que se representa con la fórmula `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



(7.5)

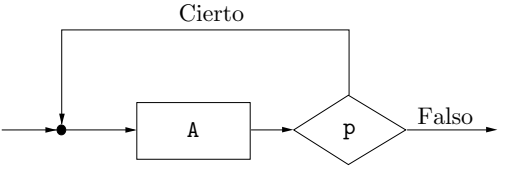
que se suele leer como:

según `s` **valga**
  `a` → **hacer** `fa`
  ⋮
  `n` → **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:



(7.6)

y se denota como `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**.

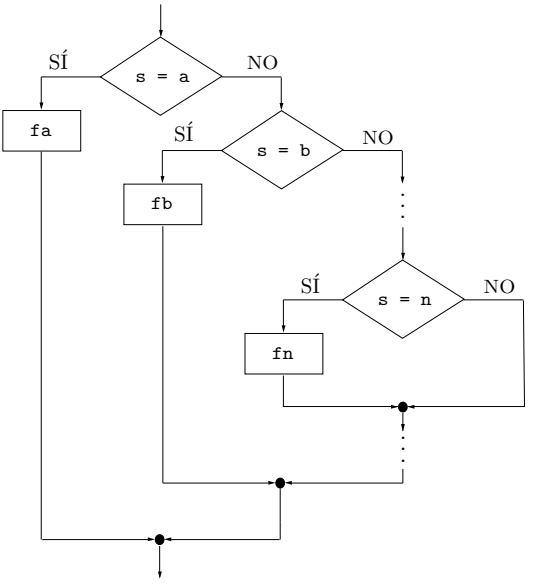


Figura 7.7. Expresión de `CaseOf` en función de `IfThenElse`.

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 seudocódigo; 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.

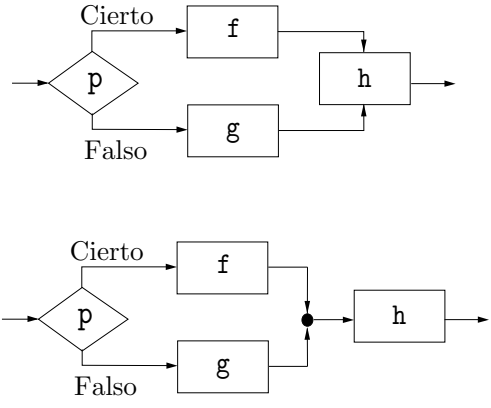
Como ejemplo, considérense los diagramas de las figuras 7.2 y 7.3: no resulta difícil comprobar que, para una misma entrada de datos, los cálculos especificados por cada diagrama son exactamente los mismos y, por lo tanto, ambos diagramas son equivalentes.

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.

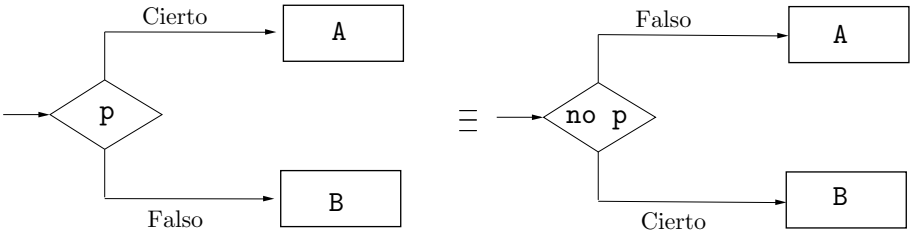
Para realizar esta transformación se pueden usar, entre otras, las operaciones de agrupamiento, inversión de predicados y desdoblamiento de bucles. A continuación se describirán estas operaciones y veremos algunos ejemplos que aclararán estos conceptos.

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

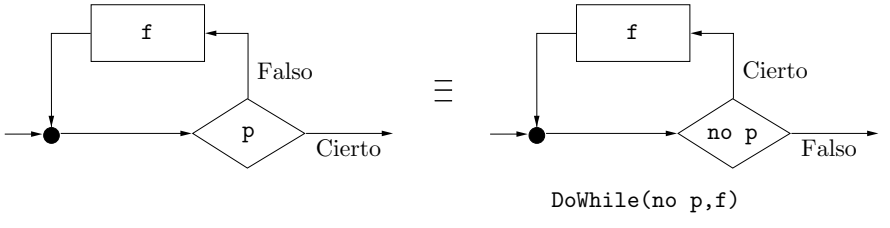
agrupamiento tras una selección.



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



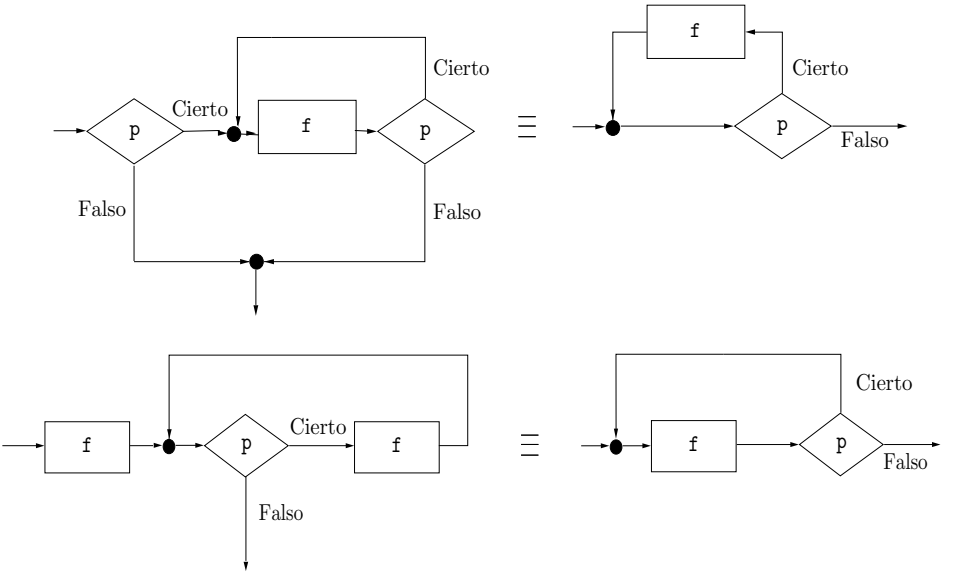
Esta transformación es especialmente útil en casos de iteración: a veces es necesario invertir el bucle para que aquélla se realice sólo cuando la condición es cierta.[2] Por ejemplo, el diagrama de la izquierda se puede modificar mediante la inversión del predicado `p` y expresarlo como aparece a la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



que es un diagrama estructurado del tipo `DoWhile(no p,f)`.

[2]Véase 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 clase[3] 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.

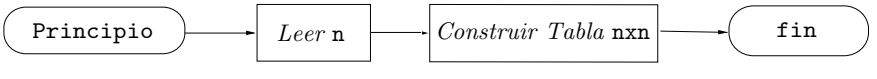
A continuación se muestran algunos ejemplos de refinamiento progresivo:

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

$$\begin{array}{cccccc} 1 & 2 & 3 & \cdots & 10 \\ 2 & 4 & 6 & \cdots & 20 \\ 3 & 6 & 9 & \cdots & 30 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 10 & 20 & 30 & \cdots & 100 \end{array}$$

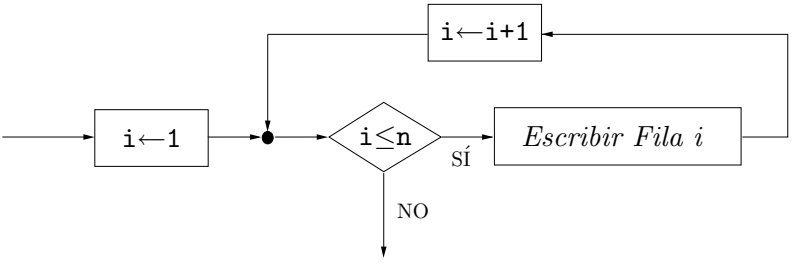
Una primera versión (burda) del programa podría ser



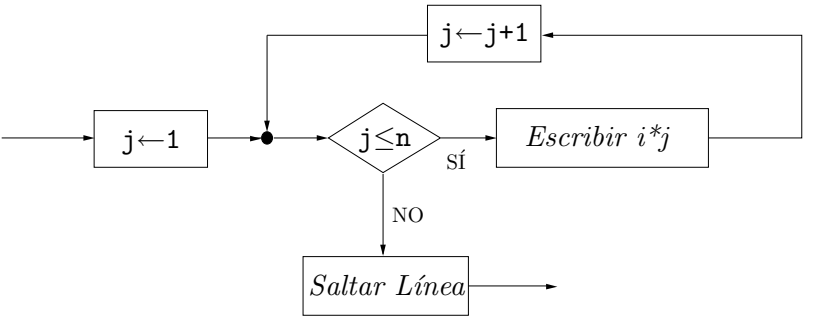
[3]... y se dispone de un buen profesor...

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 lleguemos a los múltiplos de `n`. Ésta es la idea subyacente al siguiente refinamiento de la función que construye la tabla de tamaño `n` × `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 seudocódigo, de modo que al llegar al último refinamiento la traducción a un lenguaje de programación sea prácticamente inmediata.

La primera aproximación en seudocódigo 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←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←1 hasta n hacer
    {Escribir la línea i-ésima:}
  para j←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 seudocódigo simplemente se descompone la tarea encomendada en tres subtareas: 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 seudocódigo 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,

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

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

[4]El 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,

```
Sqr(Round(SqRt(N))) = 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,

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

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
```

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

```
  {m = mín k ∈ ℕ 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,
$$m = \text{ mín } \{k \in \mathbb{N} \text{ tal que } k^2 \geq N\}$$
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 = mín k ∈ ℕ tal que k² ≥ 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 $k \in \mathbb{N}$ tal que $k^2 \geq N$ consiste en tantear esa condición sucesivamente para los valores de $i = 0, 1, \ldots$ 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, `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 $\{0, \ldots, N\}$ 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 $\{izda, \ldots, dcha\}$, *manteniendo* $izda \leq m \leq dcha$

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 método[6] 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:

[6] Y 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 $izda \leq dcha$:

$$\{izda \leq m \leq dcha \text{ y } izda \leq c < dcha \text{ y } c^2 < N\}$$
```
  izda:= c + 1
```
$$\{izda \leq m \leq dcha\}$$

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 seudocódigo 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, $1 + 2$, $1 + 2 + 3$,... Use seudocódigo y la técnica de diseño descendente para desarrollar programas que:

   (a) calculen el n-ésimo número triangular,

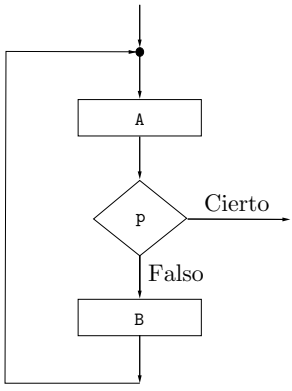


Figura 7.8.

(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

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

   multiplicando los 50 primeros factores.

(b) mediante la fórmula de Leibnitz

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

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

(c) mediante la fórmula de Vieta

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

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 $\text{sen}(\frac{\pi}{6})$, mediante su desarrollo de Taylor

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

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

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 refinamiento[1] 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)
```

[1]En 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.

[2]En 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
```

[3]Por 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 $2 * 3 * ... * n$ 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;
```

[4]En 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`.

[5] En 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.

[6] En 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 EscribirSiguiente (v: integer);
  {Efecto:  escribe en la pantalla v + 1}
begin
  v:= v + 1;
  WriteLn(v)
end; {EscribirSiguiente}
```

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

```
w:= 5;                    |
WriteLn(w);               | 5
EscribirSiguiente(w);     | 6
WriteLn(w)                | 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);                    || 5
IncrementarYescribir(w);       || 6
WriteLn(w)                     || 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.

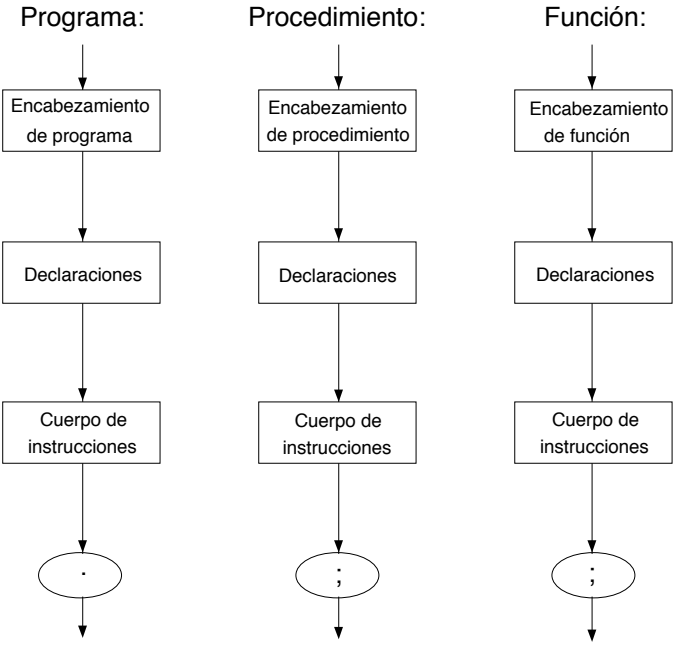


Figura 8.1. Diagramas sintácticos generales de programa, procedimiento y función.

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-

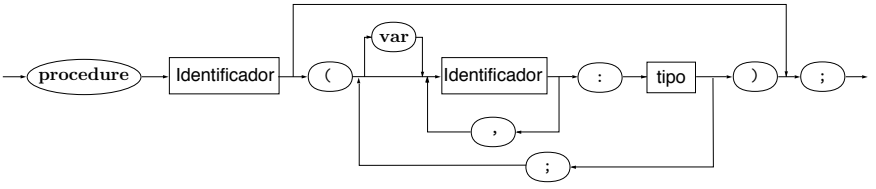


Figura 8.2. Diagrama sintáctico del encabezamiento de procedimiento.

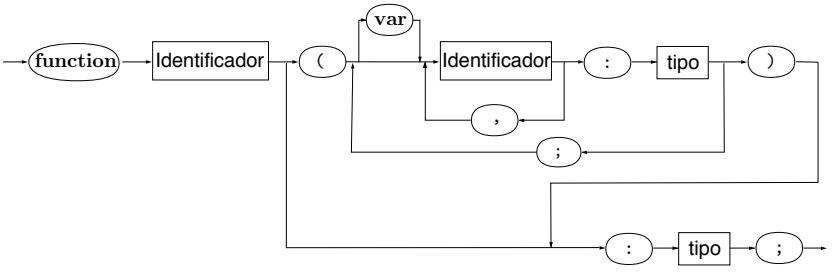


Figura 8.3. Diagrama sintáctico del encabezamiento de función.

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;
```

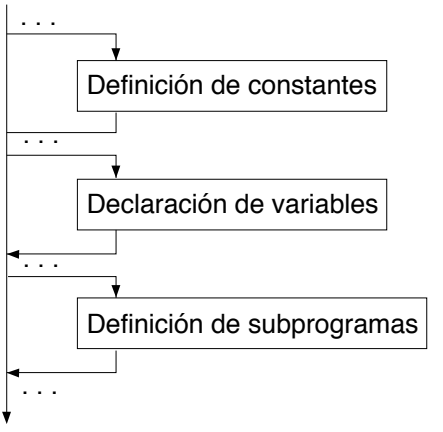


Figura 8.4. Diagrama sintáctico de las declaraciones y definiciones.

```
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}).

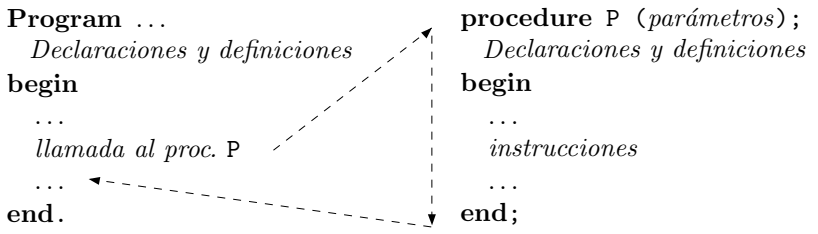


Figura 8.5. Llamada a un subprograma.

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

Estado 1:

num
?

Estado 2A, al principio de la llamada a `LeerNumPos`:

num	n
?	?

Estado 2B, al final de la llamada a `LeerNumPos`:

num	n
5	5

Estado 3, tras la llamada a `LeerNumPos` en el programa principal:

num
5

Estado 4A, al comienzo de la llamada a `Fac`:

num	i	prodAcum	Fac
5	?	?	?

Estado 4B, al comienzo de la llamada a `Fac`:

num	i	prodAcum	Fac
5	6	120	120

Estado 5, al terminar el programa principal:

num
5

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.

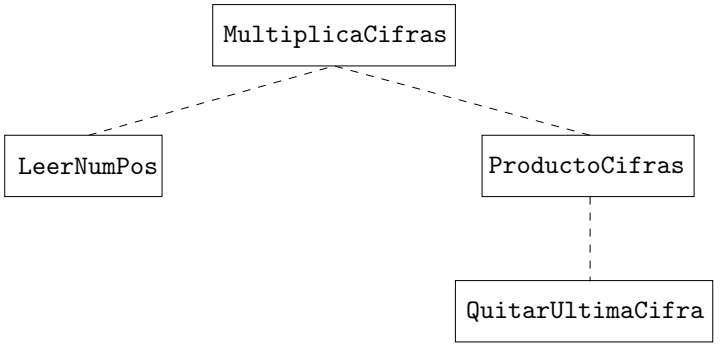


Figura 8.6.

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]

[7]Es 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 ≤ n ≤ 7 y n ≤ MaxInt}
  {Devuelve 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}
```

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! ≤ MaxInt, ya que 8! = 40320 > 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

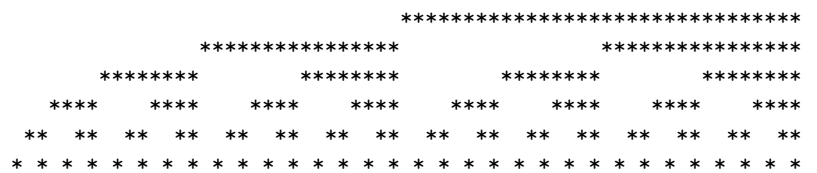
(c) una letra, haciendo uso de las funciones anteriores

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),

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

   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:

$$\textit{dividendo} \mid \textit{divisor}$$

   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:

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

[9] A esta cantidad se le llama raíz digital.

[10] Un 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

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 = máximo común divisor de n y d
Calcular $n' = n$ **div** *mcd*
Calcular $d' = d$ **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**
  $r \leftarrow n$ **mod** 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`.

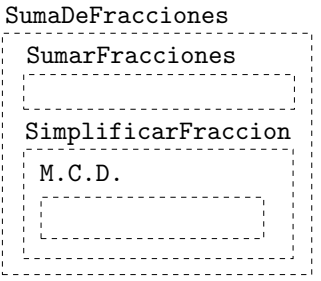


Figura 9.1. Estructura de bloques de `SumaDeFracciones`.

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.

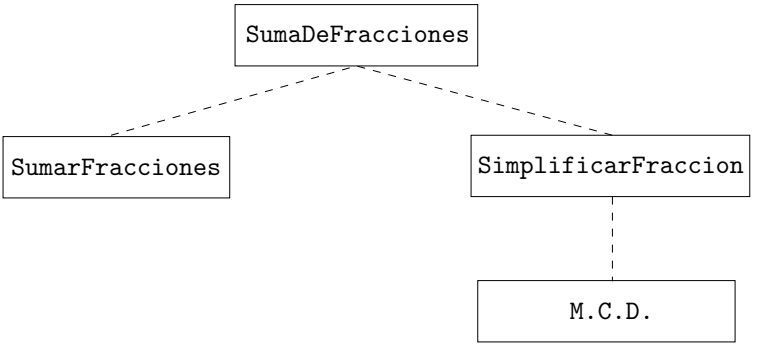


Figura 9.2. Estructura jerárquica del programa.

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]

[1]Aunque 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.

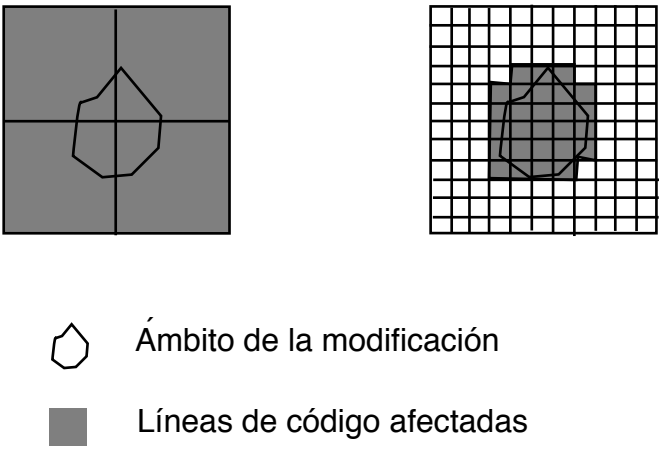


Figura 9.3.

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 ≠ 0 y d ≠ 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 seleccion 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.

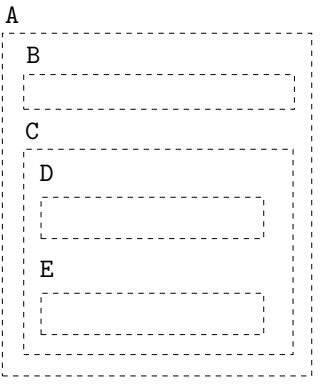


Figura 9.4. Subordinación de bloques.

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.

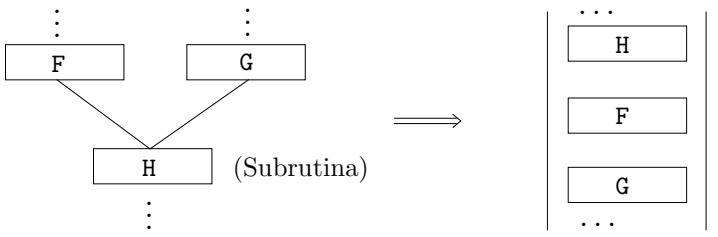


Figura 9.5.

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.

[2]Estas 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 simplificadamente 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:

$$[x_{mínima}, x_{máxima}] \times [y_{mínima}, y_{máxima}]$$

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 $númX \times númY$ 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 = 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`, ...) 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 `<` ó `>`, según caiga por la izquierda o por la derecha, respectivamente.

Así pues, el programa consta de cuatro pasos:

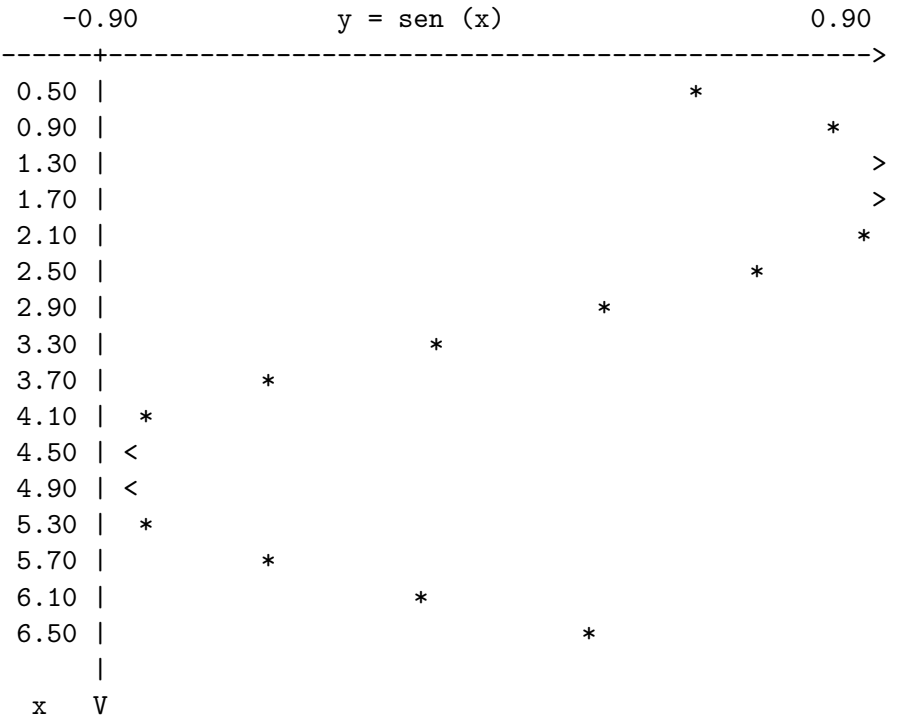

-0.90
y = sen (x)
0.90
0.50
0.90
1.30
1.70
2.10
2.50
2.90
3.30
3.70
4.10
4.50
4.90
5.30
5.70
6.10
6.50
x
V

Figura 9.6.

Pedir los datos $x_{mínima}, x_{máxima}, y_{mínima}, y_{máxima}$
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:

$$x_i = x_{mín} + i\frac{x_{máx} - x_{mín}}{\texttt{NumX}}, \quad i \in \{0, \ldots, \texttt{NumX}\}$$

Para cada ordenada $y_i = f(x_i)$, su posición (que será un entero de $\{0, \ldots \texttt{NumY}\}$ cuando $y_i \in [y_{mí}, y_{máx}]$:

$$[y_{mín}, y_{máx}] \rightarrow \{0, \ldots, \texttt{NumY}\}$$

Ello se consigue sencillamente así:

$$posY_i = \texttt{Round}\left(\texttt{NumY}\frac{y_i - y_{mín}}{y_{máx} - y_{mín}}\right)$$

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:

[3]Dadas 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:

   si $a>b$, *entonces* $m.c.d.(a,\ b) = m.c.d.(a-b,\ b)$
   si $a<b$, *entonces* $m.c.d.(a,\ b) = m.c.d.(a,\ b-a)$
   si $a=b$, *entonces* $m.c.d.(a,\ b) = m.c.d.(b,\ a) = a = b$

   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. Organíce los subprogramas de acuerdo con el siguiente diagrama de la figura 9.7.[4]

[4]Obsérvese 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`.)

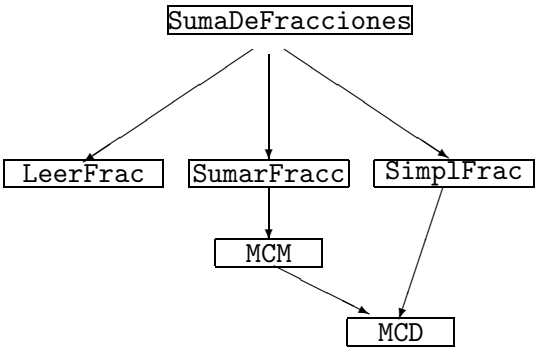


Figura 9.7.

4. Desarrolle una función `Producto`

   ```
   Producto(a, b) = a * (a+1) * ... * (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:

   $$\begin{array}{ccccccccccc} & & & & 1 & & & & & & \\ & & & 1 & & 1 & & & & & \\ & & 1 & & 2 & & 1 & & & & \\ & 1 & & 3 & & 3 & & 1 & & & \\ 1 & & 4 & & 6 & & 4 & & 1 & & \\ & & & & \ldots & & & & & & \end{array}$$

   hasta la línea `numLinea`, dato extraído del `input`.

5. Defina distintas versiones de la función *arcsen* según las siguientes descripciones:

   (a) $arcsen(x) = arctg\frac{x}{\sqrt{1-x^2}}$

   (b) $arcsen(x) = x + \frac{1}{2}\frac{x^3}{3} + \frac{1*3}{2*4}\frac{x^5}{5} + \frac{1*3*5}{2*4*6}\frac{x^7}{7} + \ldots$

   (c) Como $arcsen(a)$ es un cero de la función $f(x) = sen(x) - a$, 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

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:

$$n! = n * (n-1) * (n-2) * ... * 1$$

Como, a su vez,

$$(n-1)! = (n-1) * (n-2)... * 1$$

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}
```

[1]En 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,

$$n \;\rightarrow\; n-1 \;\rightarrow\; \ldots \;\rightarrow\; 1 \;\rightarrow\; 0$$

[2] La 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.

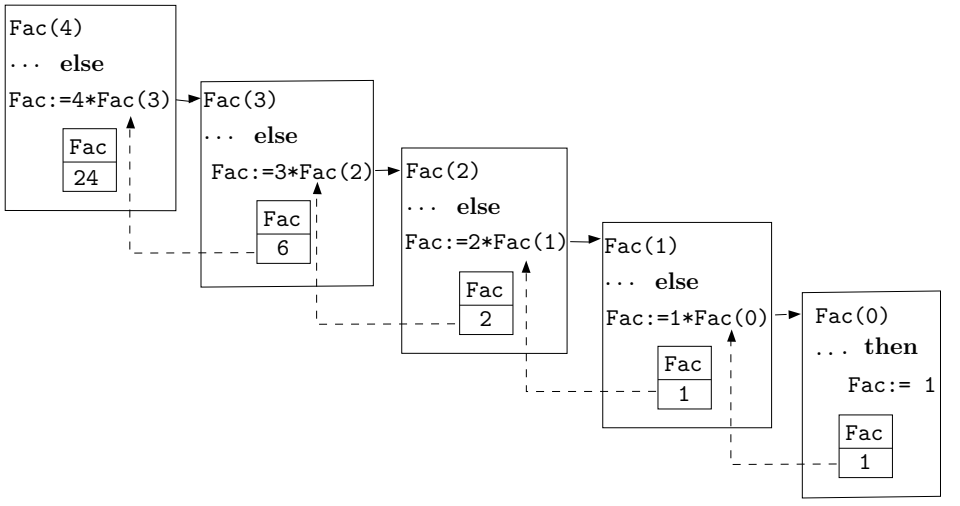


Figura 10.1. Esquema de llamadas de `Fac`.

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,

$$-3 \;\rightarrow\; -4 \;\rightarrow\; -5 \;\rightarrow\; \ldots$$

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.

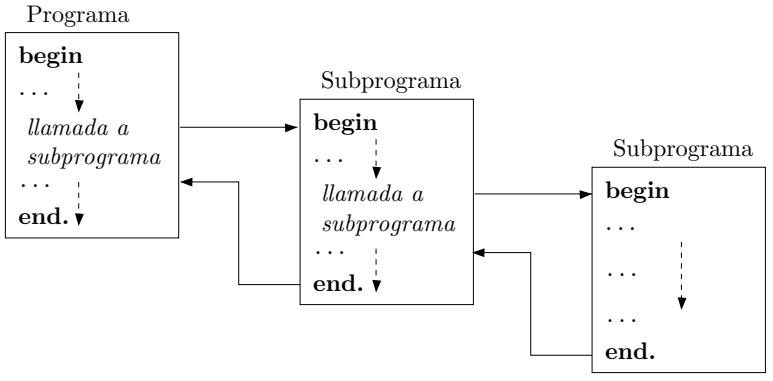


Figura 10.2. Esquema de llamadas de subprogramas.

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 enésimo 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.  fib_num}
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.

[3] Descubierta por Leonardo da Pisa (1180-1250) y publicada en su *Liber Abaci* en 1202.

[4] Según 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.

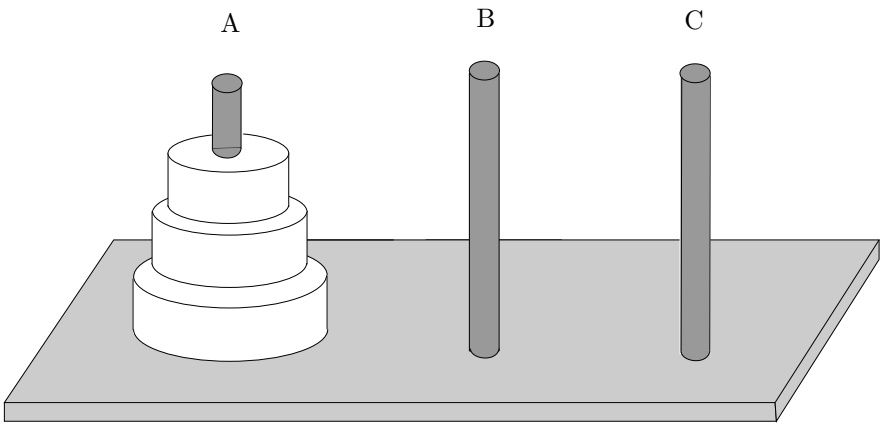


Figura 10.3. Las torres de Hanoi.

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 descanse 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:

$$\textit{Pasar dos discos de } A \textit{ a } B = \begin{cases} \textit{Mover un disco de } A \textit{ a } C \\ \textit{Mover un disco de } A \textit{ a } B \\ \textit{Mover un disco de } C \textit{ a } B \end{cases}$$

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:

$$\textit{Pasar 3 discos de } A \textit{ a } B = \begin{cases} \textit{Pasar dos de } A \textit{ a } C = \begin{cases} \textit{Mover 1 disco de } A \textit{ a } B \\ \textit{Mover 1 disco de } A \textit{ a } C \\ \textit{Mover 1 disco de } B \textit{ a } C \end{cases} \\ \textit{Mover un disco de } A \textit{ a } B \\ \textit{Pasar dos de } C \textit{ a } B = \begin{cases} \textit{Mover 1 disco de } C \textit{ a } A \\ \textit{Mover 1 disco de } C \textit{ a } B \\ \textit{Mover 1 disco de } A \textit{ a } B \end{cases} \end{cases}$$

En general, *Pasar n discos de A a B* (siendo $n \geq 1$), consiste en efectuar las siguientes operaciones,

$$\textit{Pasar n discos de } A \textit{ a } B = \begin{cases} \textit{Pasar } n{-}1 \textit{ discos de } A \textit{ a } C \\ \textit{Mover 1 disco de } A \textit{ a } B \\ \textit{Pasar } n{-}1 \textit{ discos de } C \textit{ a } B \end{cases}$$

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
```

[5]Como 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:

$$
\begin{aligned}
&Ack(0, n) = n + 1 \\
&Ack(m, 0) = Ack(m - 1, 1), \text{ si } m > 0 \\
&Ack(m, n) = Ack(m - 1, Ack(m, n - 1)) \text{ si } m, n > 0
\end{aligned}
$$

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 (aquélla 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 $E = \varnothing$, se tiene que $\mathcal{P}(E) = \varnothing$, y por tanto $N = 1 = 2^0$.

Supongamos ahora que para $n-1$, es decir, para $E = \{x_1, x_2, \ldots, x_{n-1}\}$ 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 $E = \{x_1, x_2, \ldots, x_n\}$ 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 `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

   $$\texttt{Fac}(\texttt{num} - 1) = (\texttt{num} - 1)!$$

   entonces

   $$\begin{aligned} \texttt{Fac}(\texttt{num}) &= \texttt{num} * \texttt{Fac}(\texttt{num} - 1) \\ &= \texttt{num} * (\texttt{num} - 1)! \\ &= \texttt{num}! \end{aligned}$$

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 $n, n-1, \ldots$, 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
```

[6]Se 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 caracter ']'
   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; {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) $(1 + \frac{1}{2} + \ldots + \frac{1}{n}) = (1 + \frac{1}{2} + \ldots + \frac{1}{n-1}) + \frac{1}{n}$

[7] Como es, por ejemplo, el caso de la función factorial.

(b) La potencia de un real elevado a un entero positivo:

$$\begin{array}{lcll} x^0 & = & 1 & \\ x^n & = & (x * x)^{\frac{n}{2}}, & \text{si } n > 0 \text{ y es par} \\ x^n & = & x * (x^{n-1}), & \text{si } n > 0 \text{ y es impar} \end{array}$$

(c) La cifra i-ésima de un entero n; es decir,

- la última, si $i = 0$
- la cifra (i-1)-ésima de n **div** 10, en otro caso.

(d) El coeficiente binomial, definido recurrentemente:

$$\binom{n}{0} = \binom{n}{n} = 1$$

$$\binom{n}{k} = \binom{n-1}{k} = \binom{n-1}{k-1}$$

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 $med = (inf + sup)$ **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 ($\mid b - a \mid < \epsilon$), en cuyo caso aceptamos que $\int_a^b f(x)dx \simeq (b - a) * f(m)$.

8. Sabiendo que `0` es par, es decir,

```
EsPar(0) ⇝ true
EsImpar(0) ⇝ false
```

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

Capítulo 11

Tipos de datos simples y compuestos

Ya hemos visto que los programas se describen en términos de acciones y datos. En cuanto a las acciones, se ha mostrado que admiten un tratamiento estructurado, pudiéndose combinar mediante unos pocos esquemas: la secuencia, la selección y la repetición. De igual forma se pueden estructurar los datos. Hasta ahora sólo hemos trabajado con los tipos de datos que están predefinidos en Pascal (`integer, real, char` y `boolean`), pero en muchas situaciones se manejan unidades de información que necesitan algo más que un dato predefinido, por ejemplo:

- Un color del arco iris, (rojo, naranja, amarillo, verde, azul, añil, violeta) cuya representación mediante un carácter o un entero sería forzosamente artificiosa.
- El valor de un día del mes, que en realidad no es un entero cualquiera, sino uno del intervalo [1,31], por lo que sería impreciso usar el tipo `integer`.
- El conjunto de letras necesarias para formar una cierta palabra.

- Un vector del espacio $\mathbb{R}^n$, un crucigrama o un mazo de la baraja española.
- Una ficha de un alumno donde se recoja su nombre, dirección, edad, teléfono, calificación, D.N.I. y cualquier otro tipo de información necesaria.
- Una carta.

Para poder tratar con datos como los descritos y otros muchos, Pascal permite introducir *tipos de datos definidos por el programador*.

Así, podemos clasificar los datos en dos grandes grupos:

1. Los tipos de datos *simples* que son aquéllos cuyos valores representan un dato atómico. Dentro de estos tipos de datos se encuentran los tipos predefinidos `integer, real, boolean` y `char` junto con los nuevos tipos *enumerado* y *subrango*, los cuales permiten recoger datos como, por ejemplo, los colores del arco iris y los días del mes, respectivamente.

2. Los tipos de datos *compuestos* que son aquéllos cuyos valores pueden englobar a varios datos simultáneamente. Los tipos de datos compuestos son: el tipo *conjunto* (que permite expresar el caso del conjunto de letras de una palabra), el tipo *array*[1] (que recoge los ejemplos de vectores, crucigramas o una baraja), el tipo *registro* (con cuyos valores se pueden representar fichas de alumnos), y el tipo *archivo* (en uno de cuyos valores se puede almacenar una carta).

Una vez vista la necesidad de ampliar la gama de tipos de datos disponibles, vamos a estudiar cada uno de los tipos de datos definidos por el programador (tanto los simples como los compuestos) mencionados anteriormente. Como en todo tipo de datos, tendremos que precisar su dominio (los valores que pertenecen a él) y las operaciones sobre éste.

En este capítulo se van a estudiar los tipos de datos más sencillos (*enumerado, subrango* y *conjunto*) y, junto con ellos, algunos conceptos generales, válidos para todos los tipos de datos definidos por el programador.

11.1 Tipos ordinales definidos por el programador

Los tipos de datos simples que el programador puede definir son los tipos *enumerado* y *subrango*. Éstos, junto con los tipos de datos predefinidos, son la base para construir los tipos de datos compuestos.

[1]No existe una traducción clara al castellano del término *array*, por lo que seguiremos usando este nombre en lo que sigue.

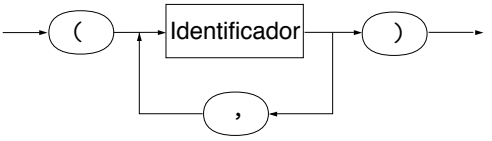


Figura 11.1.

Veamos un ejemplo en el que señalaremos la utilidad y necesidad de ampliar los tipos de datos predefinidos con estos dos nuevos tipos de datos: supongamos que deseamos hacer un horario de estudio para todos los días de la semana durante todo un año. En este caso será útil definir tipos de datos que contengan los meses (al que llamaremos tipo `tMeses`), los días de cada mes (tipo `tDiasMes`) y los días de la semana (tipo `tDiasSemana`) para poder hacer planes del tipo:

Miércoles 1 de Junio: estudiar los temas 12 y 13 de Matemáticas

Nuestro interés es definir los tipos `tDiasSemana` y `tMeses` enumerando uno a uno todos sus posibles valores y, limitar el rango del tipo `integer` a los números enteros comprendidos entre `1` y `31` para definir el tipo `tDiasMes`. Veamos que Pascal permite crear estos tipos de una forma muy cómoda.

11.1.1 Tipos enumerados

Un problema puede precisar de un determinado tipo de dato cuyos valores no están definidos en Pascal (como ocurre en el ejemplo anterior con los días de la semana). Podríamos optar por numerar los valores haciendo corresponder a cada valor un número entero que lo represente (por ejemplo, identificar los días de la semana con los números del 1 al 7), pero esta solución no es muy comprensible y, además, podría fácilmente conducirnos a errores difíciles de encontrar por la posibilidad de mezclarlos con los enteros "de verdad" y la posibilidad de aplicar operaciones sin sentido. Para solucionar este problema, Pascal nos proporciona la posibilidad de definir los datos de tipo *enumerado*. Es importante señalar que una vez que definamos un tipo de datos que contenga los días de la semana, este tipo va a ser completamente distinto del tipo predefinido `integer`. El diagrama sintáctico para la descripción de un tipo enumerado aparece en la figura 11.1.

Por ejemplo, para definir un tipo compuesto por los días de la semana incluimos:

```
type
  tDiasSemana = (lun, mar, mie ,jue, vie, sab, dom);
```

que se situará antes de la declaración de variables.

👀 Obsérvese que los valores de un tipo enumerado son identificadores, y no cadenas de caracteres (es decir, no son literales de Pascal, como los descritos en el apartado 3.6).

Como se puede observar, basta con enumerar los valores del tipo uno a uno, separándolos por comas y encerrándolos entre paréntesis. A partir de este momento Pascal reconoce el identificador `tDiasSemana` como un nuevo nombre de tipo de datos del cual se pueden declarar variables:

```
var
  ayer, hoy, mannana:  tDiasSemana;
```

Como se podría esperar, el dominio de un tipo enumerado está formado por los valores incluidos en su descripción.

Para asignar valores de tipo enumerado a las variables, se usa el operador de asignación habitual:

```
ayer:= dom;
hoy:= lun;
```

Operaciones de los tipos enumerados

En todo tipo enumerado se tiene un orden establecido por la descripción del tipo y, por lo tanto, los tipos enumerados son tipos ordinales al igual que los tipos predefinidos `integer, char` y `boolean` (véase el apartado 3.6). De este modo, los operadores relacionales son aplicables a los tipos enumerados, y el resultado de su evaluación es el esperado:

`lun < mie` $\rightsquigarrow$ `True`

`jue = sab` $\rightsquigarrow$ `False`

`(mar > lun) = (lun < mar)` $\rightsquigarrow$ `True`

Además de los operadores relacionales, son también aplicables las funciones ya conocidas `Ord`, `Pred` y `Succ`:

`Succ(jue)` $\rightsquigarrow$ `vie`

`Succ(lun) = Pred(mie)` $\rightsquigarrow$ `True`

`Pred(lun)` $\rightsquigarrow$ error (ya que no existe el anterior del primer valor)

`Succ(dom)` $\rightsquigarrow$ error (ya que no existe el siguiente al último valor)

`Ord(jue)` $\rightsquigarrow$ `3`

Como en todo tipo ordinal, salvo en el caso de los enteros, el número de orden comienza siempre por cero.

Observaciones sobre los tipos enumerados

- Dado que los tipos enumerados son ordinales, se pueden utilizar como índices en instrucciones **for**:

```
var
  d: tDiasSemana;
  ...
  for d:= lun to dom do
  ...
```

- Pueden pasarse como parámetros en procedimientos y funciones. Incluso, por tratarse de un tipo simple, puede ser el resultado de una función:

```
function DiaMannana(hoy: tDiasSemana): tDiasSemana;
  {Dev. el día de la semana que sigue a hoy}
begin
  if hoy = dom then
    DiaMannana:= lun
  else
    DiaMannana:= Succ(hoy)
end; {DiaMannana}
```

- Los valores de un tipo enumerado no se pueden escribir directamente. Para resolver este problema hay que emplear un procedimiento con una instrucción **case** que escriba, según el valor de la variable del tipo enumerado, la cadena de caracteres correspondiente al identificador del valor. El código de dicho procedimiento podría ser:

```
procedure EscribirDiaSemana(dia: tDiasSemana);
  {Efecto: escribe en el output el nombre de dia}
begin
  case dia of
    lun: WriteLn('Lunes');
    mar: WriteLn('Martes');
    ...
```

Figura 11.2.

```
    dom:  WriteLn('Domingo')
  end {case}
end;  {EscribirDiaSemana}
```

De igual forma, tampoco se puede leer directamente un valor de un tipo enumerado, por lo que, en caso necesario, habrá que desarrollar una función análoga al procedimiento `EscribirDiaSemana` que lea un valor de un tipo enumerado devolviéndolo como resultado de la función.

- No se pueden repetir valores en distintas descripciones de tipos enumerados ya que, en tal caso, su tipo sería ambiguo. Así, el siguiente fragmento es erróneo:

```
type
  tAmigos = (pepe, juan, pedro, miguel);
  tEnemigos = (antonio, pedro, enrique);
```

En este caso, la ambigüedad se refleja en que no se puede saber si el valor de `Succ(pedro)` es `miguel` o `enrique` ni si `Ord(pedro)` es `1` ó `2`.

11.1.2 Tipo subrango

El tipo de datos subrango se utiliza cuando se quiere trabajar con un intervalo de un dominio ordinal ya existente, bien de un tipo predefinido, o bien de un tipo creado con anterioridad. A este dominio ordinal lo denominamos tipo base, ya que va a ser el tipo sobre el que se define el tipo subrango.

La descripción de un tipo subrango se hace según el diagrama sintáctico de la figura 11.2

Los valores de las constantes tienen que ser de un mismo tipo ordinal y son los que van a delimitar el intervalo con el que se trabajará. Tienen que estar en orden creciente, esto es, `Ord(constante1)` $\leq$ `Ord(constante2)`, para que el intervalo tenga sentido.

Como ejemplo, consideremos las siguientes descripciones:

```
type
  tNaturales = 1..MaxInt;
  tDiasMes = 1..31;
  tContador = 1..20;
```

El uso de tipos subrango es muy aconsejable debido a que:

- El compilador puede comprobar que el valor almacenado en una variable de un tipo subrango se encuentra en el intervalo de definición, produciendo el correspondiente error en caso contrario (véase el apartado C.3.3).
- Proporcionan una mayor claridad, ya que el compilador verifica la consistencia de los tipos usados, lo que obliga al programador a una disciplina de trabajo clara y natural.

Como es lógico, el dominio de un tipo subrango estará formado por la parte del dominio de su tipo base que indique el intervalo considerado en la descripción.

Operaciones del tipo subrango

Un tipo subrango hereda todas las funciones y operaciones de su tipo base, por lo tanto se permiten realizar asignaciones, comparaciones, pasar como parámetros en procedimientos y funciones e incluso ser el resultado de una función. Así, aunque el procedimiento `EscribirDiaSemana` tenía como parametro una variable de tipo `tDiasSemana`, la herencia recibida por el tipo subrango permite realizar la siguiente instrucción:

```
type
  tDiasSemana = (lun, mar, mie, jue, vie, sab, dom);
  tLaborables = lun..vie;

var
  d:  tLaborables;
...
  EscribirDiaSemana(d)
```

Observaciones sobre el tipo subrango

- El tipo base puede ser cualquier tipo ordinal, es decir, `char`, `integer`, `boolean` o un tipo enumerado definido anteriormente, como ocurre en el tipo `tLaborables` y en el siguiente ejemplo:

```
type
  tMeses = (ene, feb, mar, abr, may, jun, jul, ago, sep,
            oct, nov, dic);
  tVerano = jul..sep;
```

Los valores de un tipo subrango conservan el orden de su tipo base.

- Turbo Pascal verifica las salidas de rangos siempre que se le indique que lo haga (véase el apartado C.3.3).

11.2 Definición de tipos

Como hemos visto, Pascal permite al programador utilizar tipos de datos propios y para ello es necesario que éste defina los tipos que quiere crear. La definición de un tipo consiste, básicamente, en dar nombre al nuevo tipo y especificar cuáles serán sus valores, esto es, nombrar el tipo y definir su dominio. Una vez que se haya definido un tipo, ya podremos declarar variables no sólo de los tipos predefinidos, sino también de este nuevo tipo definido por el programador. Naturalmente, la definición de tipos debe situarse antes de la declaración de variables.

Aunque en la mayoría de los casos se puede obviar la definición de tipos, ésta es muy recomendable para desarrollar programas más legibles, claros y no redundantes. Este hecho será explicado más adelante.

Las definiciones de tipo se sitúan entre las definiciones de constantes y de variables, como se muestra en la figura 11.3. El diagrama sintáctico para la definición de tipos es el de la figura 11.4, donde la palabra reservada **type** indica el comienzo de la definición de tipos, `identificador`[2] es el nombre que deseamos ponerle y `Tipo` es la descripción del tipo que vamos a nombrar.

Por ejemplo, en el caso de los tipos simples la descripción del tipo puede ser:

1. Una enumeración de sus valores (tipos *enumerados*):

```
type
  tColores = (rojo, azul, amarillo, negro, blanco);
```

pudiendo entonces hacer declaraciones de variables como:

[2]En este texto se seguirá el convenio de anteponer la letra `t` en los identificadores de los tipos de datos.

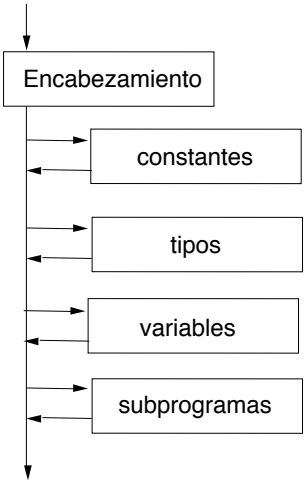


Figura 11.3.

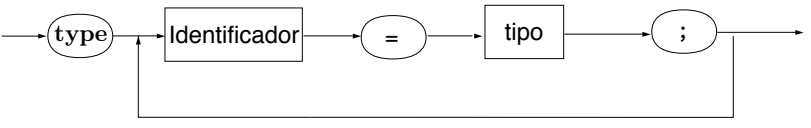


Figura 11.4.

```
var
  color:  tColores;
```

2. Un intervalo de un tipo ordinal existente, sea predefinido o enumerado (es decir, cualquier tipo *subrango*):

```
type
  tColores = (rojo, azul, amarillo, negro, blanco);
  tNatural = 0..MaxInt;
  tPrimarios = rojo..amarillo;
```

pudiendo entonces declarar variables como:

```
var
  color:  tPrimarios;
  contador:  tNatural;
```

Como veremos más adelante, para poder definir tipos compuestos, se usarán palabras reservadas (tales como **set**, **array**, **record** o **file**) para los tipos de datos *conjunto, array, registro* y *archivo* respectivamente.

11.2.1 Observaciones sobre la definición de tipos

Además de las características señaladas anteriormente para las definiciones de tipos, podemos hacer las siguientes observaciones:

1. Hasta ahora, hemos estado utilizando variables de los tipos predefinidos o bien de tipos definidos anteriormente con una instrucción **type**. Sin embargo, existe otra forma de declarar variables de tipo enumerado o subrango sin necesidad de definir el tipo previamente. Para ello se incluye directamente la descripción del tipo enumerado o subrango en la zona de declaración de variables. Por ejemplo:

```
var
  dia :  (lun, mar, mie, jue, vie, sab, dom);
  diasMes :  1..31;
```

Esta forma de definición de tipo recibe el nombre de *tipos anónimos* y las variables declaradas así reciben el nombre de *variables de tipo anónimo*. En cualquier caso, su utilización no es recomendable, sobre todo si se van a utilizar varias variables de un mismo tipo anónimo en distintos subprogramas del mismo programa, ya que habría que definir el tipo cada vez que necesitemos alguna variable local de estos tipos.

2. Se pueden renombrar los tipos predefinidos en una definición de tipos:

```
type
  tEntero = integer;
```

   El renombramiento, aunque no es recomendable, puede ser útil para evitar memorizar algún tipo predefinido; así, en el ejemplo anterior, podremos hacer declaraciones de variables de tipo `tEntero` y no necesitaremos recordar que en Pascal sería `integer`:

```
var
  n, i, j :  tEntero;
```

3. Pascal permite redefinir los tipos predefinidos:

```
type
  boolean = (falso, verdadero);
```

   De todas formas, no es recomendable redefinir un tipo predefinido, ya que el código resultante puede ser confuso y difícil de entender o modificar por otra persona distinta de la que lo ha escrito.

4. No se pueden redefinir palabras reservadas como valores de un tipo enumerado ni como nombres de tipo:

```
type
  notas = (do, re, mi, fa, sol, la, si);
  while = (nada, poco, bastante, mucho);
```

   En el ejemplo anterior, la primera definición de tipo enumerado es incorrecta porque uno de sus valores (**do**) es una palabra reservada; tampoco es posible hacer la segunda definición porque el nombre (**while**) utilizado para el tipo es una palabra reservada.

5. Es muy importante, como se ha comentado anteriormente, la elección del identificador de tipo a la hora de definir un tipo subrango o enumerado de forma que permita identificar claramente lo que queremos definir.

6. No se puede poner la definición de un tipo (anónimo) en el encabezamiento de un subprograma, por lo que el siguiente encabezamiento sería incorrecto:

```
function MannanaMes(d:  1..31) :  1..31;
```

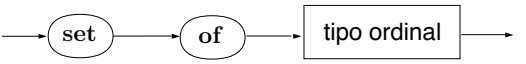


Figura 11.5.

Es preciso definir previamente el tipo y hacer referencia a su identificador en el encabezamiento, como se muestra a continuación:

```
type
  tDiaMes = 1..31;

function MannanaMes(d:  tDiaMes):  tDiaMes;
```

11.3 Conjuntos

El primer tipo de datos compuesto que vamos a estudiar en Pascal es el tipo *conjunto*, que intenta representar el concepto de los conjuntos utilizados en Matemáticas. Así, podemos definir un conjunto como una colección de objetos de un tipo ordinal. En efecto, los elementos de un conjunto no ocupan en él una posición determinada;[3] simplemente, se puede decir que pertenecen o no al mismo.

El tipo de los elementos que integran el conjunto se llama *tipo base*, que en Pascal debe ser ordinal (véase el apartado 3.6). La definición de un tipo de datos conjunto se hace según el diagrama sintáctico de la figura 11.5.

No hay que confundir los elementos de un conjunto `C` con el dominio del tipo **set of** `C`, que sería $\mathcal{P}(\texttt{C})$, es decir el conjunto formado por todos los posibles subconjuntos de `C`.

El cardinal máximo de un conjunto en Pascal depende del compilador que estemos utilizando. Por ejemplo, en Turbo Pascal se admiten conjuntos de hasta 256 elementos (con ordinales comprendidos entre 0 y 255), por lo que no podremos definir:

```
type
  tConjunto1 = set of 1..2000;
  tConjunto2 = set of integer;
```

[3]Nosotros vamos a considerar los conjuntos como datos compuestos (por ninguno o más elementos) aunque sin estructura, por carecer de organización entre sus elementos. Sin embargo, algunos autores clasifican los conjuntos como un dato estructurado.

En nuestros programas utilizaremos conjuntos con una cardinalidad menor. Por ejemplo:

```
type
  tConjuntoCar = set of char;
```

Una vez definido, podemos declarar variables de dicho tipo:

```
var
  vocales, letras, numeros, simbolos, vacio:  tConjuntoCar;
```

Para asignar valores a un conjunto se utiliza la instrucción usual de asignación, representándose los elementos del conjunto entre corchetes. Dentro del conjunto podemos expresar los valores uno a uno o indicando un intervalo abreviadamente (con una notación similar a la utilizada para definir un tipo subrango):

```
vocales:= ['A','E','I','O','U'];
letras:= ['A'..'Z'];
simbolos:= ['#'..'&','?'];
vacio:= []
```

11.3.1 Operaciones sobre el tipo conjunto

Las operaciones que se pueden realizar con los conjuntos son las que normalmente se utilizan con los conjuntos matemáticos, es decir: unión, intersección, diferencia, igualdad, desigualdad, inclusión y pertenencia. Pasamos ahora a ver la descripción y efecto de cada una de ellas:

1. *Unión* de conjuntos ($\cup$): se expresa con el signo `+`, y su resultado es el conjunto formado por todos los elementos que pertenecen al menos a uno de los conjuntos dados:

   `['A'..'C'] + ['B'..'D','G'] + ['A','E','I','O','U']` $\rightsquigarrow$
   `['A','B','C','D','E','G','I','O','U']`

2. *Intersección* de conjuntos ($\cap$): se expresa con el signo `*`, y su resultado es el conjunto formado por los elementos comunes a todos los conjuntos dados:

   `['A'..'C'] * ['B'..'F']` $\rightsquigarrow$ `['B','C']`

3. *Diferencia* de conjuntos ($\setminus$): se expresa con el signo `-`, y su resultado es el conjunto formado por los elementos que pertenecen al primer conjunto y no pertenecen al segundo:

   `['A'..'C'] - ['B'..'F']` $\rightsquigarrow$ `['A']`

4. *Igualdad* de conjuntos (=): se utiliza el símbolo `=` y representa la relación de igualdad de conjuntos (iguales elementos):

   `['A'..'D'] = ['A','B','C','D']` $\rightsquigarrow$ `True`

5. *Desigualdad* de conjuntos ($\neq$): se utiliza el símbolo `<>` que representa la relación de desigualdad de conjuntos:

   `['A'..'D'] <> ['A','B','C','D']` $\rightsquigarrow$ `False`

6. *Inclusión* de conjuntos ($\subseteq$, $\supseteq$): se utilizan dos símbolos:

   - El símbolo `<=` que representa relación de inclusión *contenido en*:
     `['A'..'D'] <= ['A','C','D','E']` $\rightsquigarrow$ `False`
   - El símbolo `>=` que representa la relación de inclusión *contiene a*:
     `['A'..'Z'] >= ['A','H','C']` $\rightsquigarrow$ `True`

7. *Pertenencia* ($\in$): se utiliza para saber si un elemento pertenece a un conjunto. Para ello se utiliza la palabra reservada **in**:

   `'A' in ['A'..'D'] * ['A','D','F']` $\rightsquigarrow$ `True`

Las operaciones unión, intersección, diferencia y los operadores relacionales y el de pertenencia se pueden combinar en una misma sentencia, en cuyo caso se mantienen las reglas de prioridad descritas en el apartado 3.5. Estas prioridades se pueden cambiar mediante el uso de paréntesis:

`['A'..'D'] + ['A'..'C'] * ['B'..'H']` $\rightsquigarrow$ `['A'..'D']`

mientras que

`(['A'..'D'] + ['A'..'C']) * ['B'..'H']` $\rightsquigarrow$ `['B'..'D']`

Un ejemplo típico de aplicación de conjuntos es simplificar las condiciones de un **repeat** o un **if**:

```
type
  tDiasSemana = (lun, mar, mie, jue, vie, sab, dom);
procedure LeerDiaSemana(var dia:  tDiasSemana);
  {Efecto:  dia es el día de la semana cuya inicial se
  ha leido en el input}
  var
    inicial :  char;
begin
  repeat
    Write('Escriba la inicial del día de la semana:  ');
    ReadLn(inicial)
  until inicial in ['L','l','M','m','X','x','J','j','V','v',
                    'S','s','D','d'];
```

```
  case inicial of
    'L','l':  dia:= lun;
    'M','m':  dia:= mar;
    'X','x':  dia:= mie;
    'J','j':  dia:= jue;
    'V','v':  dia:= vie;
    'S','s':  dia:= sab;
    'D','d':  dia:= dom
  end {case}
end;  {LeerDiaSemana}
```

Piense el lector cómo se complicaría el código de este procedimiento en caso de no usar conjuntos.

11.3.2 Observaciones sobre el tipo conjunto

Además de lo anteriormente dicho sobre el tipo de datos conjunto, cabe considerar las siguientes observaciones:

1. Podemos definir un conjunto de forma anónima, en el sentido explicado anteriormente:

```
var
  vocales, letras, numeros, simbolos, vacio:  set of char;
```

2. Al igual que para el tipo de datos enumerado, los conjuntos no se pueden leer o escribir directamente, por lo que se tendrán que desarrollar procedimientos para tal fin como los que se muestran a continuación:

```
type
  tLetrasMayusculas = 'A'..'Z';
  tConjuntoLetras = set of tLetrasMayusculas;

procedure EscribirConjunto(letras:  tConjuntoLetras);
  {Efecto:  se muestran en la pantalla los elementos
   del conjunto letras}
  var
    car:  char;
begin
  for car:= 'A' to'Z' do
    if car in letras then
      Write(car,' ')
end;  {EscribirConjunto}
```

```
procedure LeerConjunto(var conj: tConjuntoLetras);
  {Efecto: conj contiene las letras leídas del input}
  var
    car: char;
begin
  conj:= [];
  WriteLn('Escribe las letras que forman el conjunto: ');
  while not EoLn do begin
    Read(car);
    if car in ['A'..'Z']
      then conj:= conj + [car]
  end; {while}
  ReadLn
end; {LeerConjunto}
```

El primer procedimiento muestra en la pantalla todos los elementos de un conjunto formado por letras mayúsculas. Para ello recorre los valores del tipo base y comprueba la pertenencia de cada uno de ellos al conjunto, escribiéndolo en su caso.

El segundo procedimiento lee del `input` los elementos de un conjunto de letras mayúsculas. Para ello recorre los caracteres de una línea (hasta la marca ↵), incluyendo en el conjunto los del tipo base (`'A'..'Z'`) e ignorando los demás.

11.3.3 Un ejemplo de aplicación

Es conveniente concretar todas estas ideas mediante un ejemplo. En este caso vamos a utilizar el tipo de datos conjunto para implementar un programa que escriba los números primos menores que 256 (esta cota viene dada por la limitación del cardinal de un conjunto en Pascal) usando el conocido método de la *criba de Eratóstenes*.[4] La idea de esta implementación es disponer de un conjunto inicial con todos los enteros positivos menores que 256 e ir eliminando del conjunto aquellos números que se vaya sabiendo que no son primos (por ser múltiplos de otros menores). De acuerdo con esta descripción, una primera etapa de diseño del programa podría ser:

Calcular los números primos menores que 256
Escribir los números primos menores que 256

[4]Eratóstenes, conocido astrónomo y geógrafo griego (siglo III a. C.), es reconocido por haber ideado este método para elaborar una tabla de números primos. El nombre de *criba* se debe a que el algoritmo va eliminando los números que son múltiplos de otros menores, siendo los primos aquellos que quedan tras realizar esta criba.

Y en un segundo refinamiento de *Calcular los números primos menores que 256* se obtendría:[5]

Generar el conjunto inicial `primos`
para cada elemento `elem` *mayor que 1 y menor o igual que 16*
    Eliminar del conjunto todos sus múltiplos

Y en un nivel de refinamiento inferior, *Eliminar del conjunto todos sus múltiplos* se puede desarrollar así:

Dar valor inicial (igual a 2) al coeficiente `k`
repetir
  Eliminar `elem * k` *del conjunto* `primos`
  Incrementar `k` *en una unidad*
hasta que `elem * k` *sea mayor o igual que 256*

Desde este nivel de refinamiento es fácil pasar a una implementación en Pascal, como puede ser la siguiente:

```
Program Criba (output);
  {Halla y escribe los primeros números primos}
  const
    N = 255;
  type
    tConjuntoPositivos = set of 1..N;
  var
    primos:  tConjuntoPositivos;
    elem, k:  integer;

  procedure EscribirConjuntoPositivos(c:  tConjuntoPositivos);
    {Efecto:  se muestran en la pantalla los elementos
     del conjunto c}
    var
      i:  integer;
  begin
    for i:= 1 to N do
      if i in c  then
        WriteLn(i :  3, ' es primo')
  end;  {EscribirConjuntoPositivos}
```

[5]Como se comentó en el apartado 8.2.1, basta con eliminar los múltiplos de los números naturales menores o iguales que la raíz cuadrada de la cota (en nuestro ejemplo, los menores o iguales que $\sqrt{256} = 16$).

```
begin {Criba}
   primos:= [1..N];
     {Se genera el conjunto inicial}
   elem:= 2;
   while elem < SqRt(N) do begin
       {Inv.:  Para todo i ∈ primos tal que 1 ≤ i < elem,
       se tiene que i es primo}
     if elem in primos then begin
       k:= 2;
       repeat
         primos:= primos - [elem * k];
           {Se eliminan los números no primos del conjunto}
         k:= k + 1
       until elem * k > N
     end; {if}
     elem:= elem + 1
   end; {while}
   WriteLn('Los primos menores que ', N,' son');
   EscribirConjuntoPositivos(primos)
end.  {Criba}
```

11.4 Ejercicios

1. Se desea averiguar qué letras intervienen en un texto (sin distinguir entre letras mayúsculas y minúsculas ni importar su frecuencia), y cuáles se han omitido.

   (a) Defina un tipo apropiado para controlar las letras que van apareciendo.

   (b) Escriba un subprograma que lea el `input`, formado por varias líneas, ofreciendo como resultado el conjunto de las letras encontradas.

   (c) Escriba un subprograma de escritura de conjuntos de letras.

   (d) Escriba un subprograma que averigüe el cardinal de un conjunto de letras.

   (e) Integre los apartados anteriores en un programa que averigüe cuáles y cuántas son las letras distintas encontradas y las omitidas en el `input`.

2. Dados los conjuntos $conjNum \subset \{1, \ldots, 25\}$ y $conjLet \subset \{'\texttt{A}', \ldots, '\texttt{Z}'\}$, escriba un subprograma que escriba en la pantalla el producto cartesiano $conjNum \times conjLet$.

3. Un modo de calcular el máximo común divisor de dos números `m` y `n`, enteros estrictamente positivos no superiores a 50, consiste en hallar los conjuntos de sus divisores, `divisDeM` y `divisDeN`, luego su intersección, `divisComunes`, y después el mayor elemento de este conjunto.

   Desarrolle un programa que realice estos pasos dando una salida detallada del proceso seguido.

4. Los meses del año se conocen por su nombre, aunque a veces también se abrevian por su número. Desarrolle los siguientes apartados para ambas representaciones.

(a) Defina un tipo para representar los meses del año, así como subprogramas apropiados para su lectura, mediante sus letras iniciales, y escritura, ofreciendo siempre su nombre (o su número).

(b) Defina la función `ProximoMes`, y empléela en un programa que, a partir de un mes dado, ofrece los siguientes n meses.

(c) Defina la función `MesAnterior`, con un argumento `esteMes`, que repasa la lista de los meses posibles hasta dar con uno, `mesAnt`, de manera que se obtenga `ProximoMes(mesAnt)` $\rightsquigarrow$ `esteMes`. Ignore la existencia de la función estándar `Pred`.

(d) Integre los apartados anteriores en un programa que escriba una tabla con los meses del año, y para cada uno, su mes anterior y siguiente.

5. Escriba un programa que lea los caracteres del `input` hasta su final, desprecie los que no son letras mayúsculas y, con éstas, forme el conjunto de las que no aparecen, el de las que aparecen una vez y el de las que aparecen más de una.

6. **Partes de un conjunto**

(a) Dado un conjunto C, desarrolle un programa que escriba en la pantalla todos los posibles subconjuntos de C, esto es, el conjunto de sus partes, $\mathcal{P}(C)$.

(b) Escriba un programa para mostrar que todo conjunto C tiene 2^n subconjuntos, siendo $n = card(C)$. Para ello, modifique el programa anterior de manera que genere las partes de C y las cuente en vez de escribirlas en la pantalla.

7. **Combinaciones de un conjunto**

(a) Dado un conjunto C, de cardinal m, desarrolle un programa que escriba en la pantalla todas las posibles combinaciones que pueden formarse con n elementos de C, siendo $n \leq m$.

(b) Escriba un programa que cuente el número de combinaciones descritas en el apartado anterior.

Capítulo 12

Arrays

En el capítulo anterior, vimos la diferencia entre tipos de datos simples y compuestos. Dentro de los datos compuestos, se estudiaron los más sencillos (los conjuntos), pero, además de éstos, existen otros datos compuestos que se construyen dotando de una estructura a colecciones de datos pertenecientes a tipos más básicos, y por esto se llaman *tipos estructurados*.

En este capítulo se presenta el tipo de datos estructurado *array*, que será útil para trabajar con estructuras de datos como, por ejemplo, vectores, matrices, una sopa de letras rectangular, una tabla de multiplicar, o cualquier otro objeto que necesite una o varias dimensiones para almacenar su contenido.

12.1 Descripción del tipo de datos array

Los arrays son tipos de datos estructurados ampliamente utilizados, porque permiten manejar colecciones de objetos de un mismo tipo con acceso en tiempo constante, y también porque han demostrado constituir una herramienta de enorme utilidad.

Consideremos los siguientes ejemplos en los que veremos la necesidad y la utilidad de usar el tipo de datos array:

- Imaginemos que queremos calcular el producto escalar, $\vec{u}\cdot\vec{v}$, de dos vectores $\vec{u}$ y $\vec{v}$ de $\mathbb{R}^3$ mediante la conocida fórmula: $\vec{u}\cdot\vec{v} = u_1v_1 + u_2v_2 + u_3v_3$.

  Con los datos simples que conocemos hasta ahora, tendríamos que definir una variable para cada una de las componentes de los vectores, es decir, algo parecido a:

```
var
  u1, u2, u3, v1, v2, v3:  real;
```

  y a la hora de calcular el producto escalar tendríamos que realizar la operación:

```
prodEscalar:= u1 * v1 + u2 * v2 + u3 * v3;
```

  Para este caso sería más natural disponer de una "variable estructurada" que agrupe en un solo objeto las componentes de cada vector. El tipo array de Pascal permite resolver este problema.

- Imaginemos que una constructora acaba de finalizar un grupo de 12 bloques de pisos (numerados del 1 al 12), cada uno de los cuales tiene 7 plantas (numeradas del 1 al 7) y en cada planta hay 3 viviendas (A, B y C). Supongamos que el encargado de ventas quiere llevar un control lo más sencillo posible sobre qué viviendas se han vendido y cuáles no. Para ello, podríamos utilizar $12 \times 7 \times 3 = 252$ variables de tipo `boolean` de la forma: `bloqueiPlantajLetraX` asignándole un valor `True` para indicar que la vivienda del bloque i, planta j, letra X está vendida o bien `False` para indicar que no lo está.

  En este caso, sería mucho más cómodo utilizar algún tipo de datos estructurado para almacenar esta información de forma más compacta y manejable (por medio de instrucciones estructuradas). La estructura más adecuada sería la de una matriz tridimensional: la primera dimensión indicaría el número del bloque, la segunda dimensión la planta, y la tercera la letra de la vivienda. Así, para indicar que en el bloque 3, el piso 5ºA está vendido, asignaremos un valor `True` el elemento que ocupa la posición `[3,5,'A']` de este dato estructurado; mientras que si en el bloque 5, el 1ºC sigue estando disponible, asignaremos un valor `False` a la posición `[5,1,'C']`.

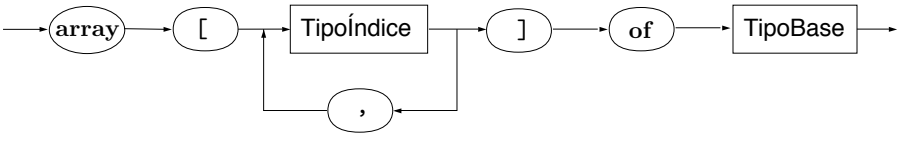
El tipo array ofrece la posibilidad de referirnos a las componentes de un modo genérico, por su posición, lo que hace más cómodo y comprensible el desarrollo de programas.

El tipo estructurado array captura la idea de los vectores y matrices del álgebra (como podrían ser $\mathbb{R}^3$, $\mathcal{M}_{2\times 5}(\mathbb{Z})$), aunque sus elementos componentes no tienen que ser números: pueden ser de cualquier tipo. A semejanza de estas estructuras matemáticas, los arrays se pueden manipular fácilmente, debido a su organización regular (es fácil verlos como hileras, tablas, estructuras cúbicas, etc.):

$$v = \begin{array}{|c|c|c|} \hline 0.24 & 3.14 & -3.56 \\ \hline \end{array} \qquad m = \begin{array}{|c|c|c|c|c|} \hline -1 & 2 & -3 & 4 & -5 \\ \hline 6 & -7 & 8 & -9 & 0 \\ \hline \end{array}$$

$$c = \begin{array}{|c|c|} \hline 'a' & 'b' \\ \hline 'c' & 'd' \\ \hline 'e' & 'f' \\ \hline \end{array}$$

El diagrama sintáctico de la definición de un array es:



Por lo tanto, su definición en Pascal es de la siguiente forma:

```
array [TipoIndice1, TipoIndice2, ..., TipoIndiceL] of TipoBase
```

o equivalentemente:

```
array [TipoIndice1] of array [TipoIndice2] of ...
  ... array [TipoIndiceL] of TipoBase
```

Los tipos `TipoIndice1, TipoIndice2 ... TipoIndiceL` tienen que ser de un tipo simple ordinal, es decir `integer`, `char`, `boolean`, enumerado o subrango. `L` es el número de dimensiones del array. Por ejemplo, se pueden realizar las siguientes definiciones de tipos y declaraciones de variables:

```
type
  tPlanetas = (mercurio, venus, tierra, marte, jupiter,
               saturno, urano, neptuno, pluton);
  tVector = array[1..3] of real;
  tMatriz = array[1..2, 1..5] of integer;
```

```
  tCubo = array[1..2, 1..2, 1..3] of char;
  tDistancia = array[tPlanetas, tPlanetas] of real;
  tUrbanizacion = array[1..12, 1..7,'A'..'C'] of boolean;
var
  u, v:  tVector;
  m:  tMatriz;
  c:  tCubo;
  d:  tDistancia;
  costaMar:  tUrbanizacion;
```

El dominio de un array es el producto cartesiano de los dominios de los tipos de los índices.

Como se ha visto en la definición genérica anterior, los arrays multidimensionales se pueden definir de varias formas (lo vemos para el caso de dos dimensiones):

1. Como un vector de vectores:

```
type
  tMatriz = array[1..8] of array['A'..'E'] of real;
```

2. Como un vector de un tipo definido anteriormente que será otro **array**:

```
type
  tVector = array['A'..'E'] of real;
  tMatriz = array[1..8] of tVector;
```

3. Introduciendo los índices dentro de los corchetes separados por comas:

```
type
  tMatriz = array[1..8, 'A'..'E'] of real;
```

Esta última es la forma más recomendable de definir un array multidimensional, ya que evita posibles errores en el orden de los índices.

Si se quiere definir una variable de tipo `tMatriz` se realiza de la forma usual y es igual para cualquiera de las tres definiciones anteriores, esto es:

```
var
  m :  tMatriz;
```

con lo que estaremos declarando una variable que será una matriz de tamaño 8×5. Otra posibilidad es la siguiente:

```
var
  m :  array[1..8, 'A'..'E'] of real;
```

12.1.1 Operaciones del tipo array y acceso a sus componentes

Las operaciones permitidas con los componentes de un array son las mismas que las permitidas a su tipo base.

Acceso y asignación a componentes de un array

Se accede a sus elementos mediante el uso de tantos índices como dimensiones tenga el array, siguiendo el siguiente esquema:

```
idArray [expres1, expres2, ..., expresL]
```

o, equivalentemente:

```
idArray [expres1][expres2]...[expresL]
```

donde `expres1, expres2, ..., expresL` son expresiones del tipo de los `L` índices de `idArray`, respectivamente. Por ser expresiones, pueden ser literales de los tipos de los índices:

```
v[3]
m[2,3] ≡ m[2][4]
c[1,2,1] ≡ c[1][2][1]
d[venus, tierra]
costaMar[12,3,'B']
```

o resultados de alguna operación; así, por ejemplo, si `i` = 2, las siguientes expresiones son equivalentes a las anteriores:

```
v[i+1]
m[i,2*i-1] ≡ m[i][2*i-1]
c[i-1,i,1] ≡ c[i-1][i][1]
d[Succ(mercurio), Pred(marte)]
costaMar[6 * i,i + 1,'B']
```

Para dar valores a las componentes de un array se usa la instrucción de asignación:

```
v[2]:= 3.14
m[i,2 * i - 1]:= 8
c[1][2][1]:= 'b'
d[mercurio, pluton]:= 3.47E38
costaMar[12,3,'B']:= True
```

Al igual que ocurre con las variables de tipo simple, una referencia a una componente de un array puede representar la posición de memoria donde se almacena su valor (como en los ejemplos anteriores), o su valor, si se utiliza dentro de una expresión (como ocurre en la instrucción `Write(v[2])`).

Asignación y operaciones de arrays completos

Además de la asignación a las componentes de un array, se pueden realizar asignaciones directas de arrays siempre que sean del mismo tipo. Por ejemplo, dadas las definiciones

```
type
  tDiasSemana = (lun, mar, mie, jue, vie, sab, dom);
  tIndice1 = -11..7;
  tIndice2 = 'A'..'Z';
  tIndice3 = lun..vie;
  tMatrizReal = array[Indice1, Indice2, Indice3] of real;
var
  i:  tIndice1;
  j:  tIndice2;
  k:  tIndice3;
  m1, m2:  tMatrizReal;
```

la asignación:

```
m2:= m1
```

es equivalente a la instrucción:

```
for i:= -11 to 7 do
    for j:= 'A' to 'Z' do
      for k:= lun to vie do
        m2[i,j,k]:= m1[i,j,k]
```

También se pueden realizar asignaciones por filas o columnas siempre que se hayan definido las filas o columnas como tipos con nombre. Por ejemplo,

```
type
  tVector = array[1..3] of real;
  tMatriz = array[1..5] of tVector;
var
  v:  tVector;
  m:  tMatriz;
  ...
  m[4]:= v
  ...
```

En Pascal no es posible comparar arrays completos aunque sean del mismo tipo. Para realizar esta operación es necesario comparar elemento a elemento, comprobando la igualdad entre ellos. Por ejemplo, suponiendo que `m1` y `m2` son matrices de tipo `tMatriz`, la comprobación de su igualdad se puede hacer así:

```
var i, j:  integer;
...
iguales := True;
i:= 0;
while iguales and (i <= 3) do begin
  j:= 0;
  while iguales and (j <= 5) do begin
    iguales := m1[i,j] = m2[i,j];
    j := Succ(j)
  end;
  i := Succ(i)
end
  {PostC.:  iguales indica si m1 = m2 o no}
```

Por otra parte, en cuanto a las operaciones de entrada y salida, solamente se pueden leer y escribir arrays completos cuando se trata de arrays de caracteres. En este caso, y por lo que respecta a la lectura, la cadena leída debe coincidir en longitud con el número de componentes del vector.

En los restantes tipos de componentes de arrays, se deben leer y escribir los valores de las componentes una a una siempre que sean de tipo simple. En el siguiente ejemplo se definen procedimientos para la lectura y escritura de un array de una dimensión:

```
const
  Tamanno = 5;
type
  tRango = 1..Tamanno;
  tVector = array[tRango] of real;
procedure LeerVector(var vec:  tVector);
  var
    i:  tRango;
begin
  for i:= 1 to Tamanno do begin
    Write('Introduzca v(',i,')= ');
    ReadLn(vec[i])
  end
end; {LeerVector}

procedure EscribirVector(vec:  vector);
  var
    i:  tRango;
begin
  for i:= 1 to Tamanno do
    WriteLn('v(',i,')= ',vec[i]);
  WriteLn
end; {EscribirVector}
```

12.1.2 Características generales de un array

Además de las operaciones permitidas con un array y la forma de acceder a sus elementos, hay que destacar las siguientes características comunes al tipo array, independientemente de su tamaño o de su dimensión:

- Los arrays son estructuras homogéneas, en el sentido de que sus elementos componentes son todos del mismo tipo.
- El tamaño del array queda fijado en la definición y no puede cambiar durante la ejecución del programa, al igual que su dimensión. Así, el tamaño de cualquier variable del tipo

  ```
  type
    vector = array[1..3] of real;
  ```

  será de 3 elementos, mientras que su dimensión será 1. Como estos valores no podrán variar, si necesitásemos utilizar un vector de $\mathbb{R}^4$ tendríamos que definir otro tipo array con tamaño 4 y dimensión 1.
- Los datos de tipo array se pueden pasar como parámetro en procedimientos y funciones, pero el tipo que devuelve una función no puede ser un array (ya que un array es un tipo de datos compuesto). Para solucionar este problema, se debe utilizar un procedimiento con un parámetro por variable de tipo array en vez de una función. Este parámetro adicional va a ser el resultado que pretendíamos devolver con la función (se incluyen ejemplos de esto en los apartados siguientes).

  Cuando los arrays son grandes y se pasan como parámetro por valor a los subprogramas, se duplica el espacio en memoria necesario para su almacenamiento, puesto que hay que guardar el array original y la copia local (véase el apartado 8.2.3). Además, el proceso de copia requiere también un cierto tiempo, tanto mayor cuanto más grande es el array. Por estos motivos, cuando el array no se modifica en el cuerpo del subprograma, es aconsejable pasarlo como parámetro por referencia, pues de esta forma no hay duplicación de espacio ni tiempo ocupado en la copia (véase el apartado 8.6.1). En cualquier caso, el programador debe cuidar extremadamente las eventuales modificaciones de esa estructura en el subprograma, ya que repercutirían en el parámetro real.

Usualmente, los arrays de una dimensión se suelen denominar *vectores,* mientras que los de más de una dimensión reciben el nombre genérico de *matrices.* En los siguientes apartados se presentan algunas de sus particularidades.

12.2 Vectores

En términos generales, un vector es una secuencia, de longitud fija, formada por elementos del mismo tipo. Teniendo en cuenta que un vector es un array de dimensión 1, su definición es sencilla. Por ejemplo:

```
type
  tNumeros = 1..10;
  tDiasSemana = (lun,mar,mie,jue,vie,sab,dom);
  tVectorDeR10 = array[tNumeros] of real;
  tFrase = array[1..30] of char;
  tVectorMuyGrande = array[integer] of real;
```

Hay que observar que el último vector llamado `vectorMuyGrande` sólo se podría definir si dispusiéramos de "mucha memoria". [1]

Con las definiciones anteriores se pueden realizar operaciones como:

```
var
  v:  tVectorDeR10;
  refran:  tFrase;

...
v[4]:= 3.141516;
v[2 * 4 - 1]:= 2.7172 * v[4];
refran:= 'Al␣que␣madruga,␣Dios␣le␣ayuda.'
...
```

Ejemplo

Veamos ahora un ejemplo de manejo de vectores en Pascal. Para indicar cuántos viajeros van en cada uno de los 15 vagones de un tren (con una capacidad máxima de 40 personas por vagón), en lugar de utilizar 15 variables enteras (una para cada vagón), se puede y se debe utilizar un vector de la siguiente forma:

```
const
  CapacidadMax = 40;
type
  tCapacidad = 0..CapacidadMax;
  tVagones = array[1..15] of tCapacidad;
var
  vagon :  tVagones;
```

[1] Por otra parte, se debe señalar que Turbo Pascal facilita un tratamiento más cómodo y directo de las cadenas de caracteres por medio de los llamados *strings* (véase el apartado B.5).

Así, haremos referencia al número de pasajeros del vagón i-ésimo mediante `vagon[i]`, mientras que el total de viajeros en el tren será

$$\sum_{i=1}^{15} \texttt{vagon[i]}$$

que en Pascal se calcula como sigue:

```
total:= 0;
for i:= 1 to 15 do
  total:= total + vagon[i]
```

Como todo array, un vector se puede pasar como parámetro en procedimientos y funciones:

```
const
  Tamanno = 5;
type
  tVector = array[1..Tamanno] of real;

function Norma(vec: tVector) : real;
  {Dev. sqrt(sum_{i=1}^{Tamanno} vec_i^2) }
  var
    i: 1..Tamanno;
    sumaCuad: real;
begin
  sumaCuad:= 0.0;
  for i:= 1 to Tamanno do
    sumaCuad:= sumaCuad + Sqr(vec[i]);
  Norma:= SqRt(sumaCuad)
end; {Norma}
```

La función anterior devuelve la norma de un vector de $\mathbb{R}^{\texttt{Tamanno}}$. Supongamos ahora que queremos desarrollar una función tal que, dado un vector, devuelva el vector unitario de su misma dirección y sentido. En este caso no podremos utilizar una función, ya que un vector no puede ser el resultado de una función. Por esta razón tendremos que utilizar un procedimiento cuyo código podría ser el siguiente:

```
procedure HallarUnitario(v: tVector; var uni: tVector);
  {PostC.: para todo i, si 1 ≤ i ≤ Tamanno, entonces uni[i] = v_i / Norma(v)}
  var
    norm: real;
    i: 1..Tamanno;
```

```
  function Norma(vec: tVector): real;
    {Dev. sqrt(sum_{i=1}^{Tamanno} vec_i^2) }
    ...
  end; {Norma}

begin
  norm:= Norma(v);
  for i:= 1 to Tamanno do
    uni[i]:= v[i]/norm
end; {HallarUnitario}
```

12.3 Matrices

Como ya se dijo, los arrays multidimensionales reciben el nombre genérico de *matrices*. En este apartado se presentan algunos ejemplos de utilización de matrices.

Evidentemente, la forma de definir los tipos de datos para las matrices es la misma de todos los arrays, así como el modo de declarar y manipular variables de estos tipos. Así, por ejemplo:

```
type
  tMeses = (ene, feb, mar, abr, may, jun, jul, ago, sep, oct,
            nov, dic);
  tDiasMes = 1..31;
  tHorasDia = 0..23;
  tFrase = array[1..30] of char;
  tMCuadrada = array[1..5, 1..5] of real;
  tFiestas95 = array[tDiasMes, tMeses] of boolean;
  tAgenda95 = array[tDiasMes, tMeses, tHorasDia] of tFrase;
var
  m: tMCuadrada;
  festivos: tFiestas95;
  agenda: tAgenda95;

begin
  m[1,2]:= SqRt(2);
  m[2,3-2]:= 2.43 * m[1,2];
  festivos[25,dic]:= True;
  festivos[28,dic]:= False;
  agenda[18,mar,17]:= 'Boda␣de␣Jose␣Luis␣y␣Mavi.␣␣␣␣␣'
  ...
end.
```

En el siguiente ejemplo, que calcula el producto de dos matrices reales, se utilizan las operaciones permitidas a los arrays, haciendo uso del hecho de que las matrices se pueden pasar como parámetros en funciones y procedimientos. En un primer nivel de diseño se tiene:

Leer matrices `a` *y* `b`
Multiplicar matrices `a` *y* `b`, *hallando la matriz producto* `prod`
Mostrar la matriz `prod`

Dada la simplicidad y familiaridad de los subprogramas a implementar, no se considera necesario un nivel inferior de diseño, razón por la cual se presenta directamente la codificación en Pascal:

```
Program MultiplicacionDeMatrices (input,output);
  {El programa lee dos matrices cuadradas de dimensión N y de
   componentes reales y las multiplica, mostrando la matriz
   producto en la pantalla}
  const
    N = 10;
  type
    tMatriz = array[1..N, 1..N] of real;
  var
    a, b, prod:  tMatriz;

  procedure LeerMatriz(var mat :  tMatriz);
    {Efecto:  Este procedimiento lee del input una matriz
     cuadrada mat ∈ M_N(IR), componente a componente}
    var
      fil, col:  1..N;
  begin
    for fil:= 1 to N do
      for col:= 1 to N do begin
        Write('Introduzca la componente ', fil, ',', col,
              ' de la matriz:  ');
        ReadLn(mat[fil,col])
      end
  end; {LeerMatriz}

  procedure MultiplicarMat(m1, m2:  tMatriz; var resul:  tMatriz);
    {Efecto:  resul:= m1 * m2}
    var
      i, j, k:  1..N;
        {i recorre las filas de m1 y de resul, j recorre las columnas
         de m1 y las filas de m2 y k recorre las columnas de m2 y las
         de resul}
```

```
  begin
    for i:= 1 to N do
      for k:= 1 to N do begin
        resul[i,k]:= 0;
        for j:= 1 to N do
          resul[i,k]:= resul[i,k] + m1[i,j] * m2[j,k]
      end {for k}
  end; {MultiplicarMat}

  procedure EscribirMatProd(m:  tMatriz);
    {Efecto:  escribe en la pantalla los elementos de la matriz m}
    var
      i, j:  1..N;
  begin
    for i:= 1 to N do
      for j:= 1 to N do
          {Escribe m_ij}
        WriteLn('m(', i, ',', j, ') = ', m[i,j]);
  end; {EscribirMatProd}

begin
  WriteLn('Lectura de la matriz A');
  LeerMatriz(a);
  WriteLn('Lectura de la matriz B');
  LeerMatriz(b);
  MultiplicarMat(a,b,prod);
  EscribirMatProd(prod)
end.  {MultiplicacionDeMatrices}
```

Un ejemplo completo

El siguiente ejemplo utiliza todos los tipos de datos definidos por el programador vistos hasta ahora, es decir, los tipos *enumerado, subrango* y array.

Se trata de construir un almanaque del siglo XX, asignando en una matriz con tres índices correspondientes a los días del mes (1 al 31), meses (ene, feb, etc.), y años desde el 1901 al 2000, los días de la semana.

El punto de partida es el día 31 de diciembre de 1900 que fue lunes. A partir de esta fecha se van recorriendo consecutivamente todos los días del siglo asignándoles el día de la semana correspondiente, teniendo especial cuidado en determinar cuántos días tiene cada mes, para lo cual se ha de comprobar si el año es bisiesto. En un primer nivel de refinamiento podríamos escribir el siguiente seudocódigo:

El día 31 de diciembre de 1900 fue lunes
Para todos los años del siglo hacer
  Para todos los meses del año hacer
    Asignar el día de la semana a cada día del mes

donde, la acción *Asignar el día de la semana a cada día del mes* puede ser refinada en un nivel inferior de la siguiente forma:

Calcular cuántos días tiene el mes
Para todos los días del mes hacer
  asignar al día actual el mañana de ayer
  asignar a ayer el día actual

Los bucles correspondientes a los años y los meses no presentan dificultad, sin embargo, los días del mes dependen del propio mes, y en el caso de febrero, de que el año sea o no bisiesto. Por eso hay que calcular `CuantosDias` tiene el mes y para ello es necesario conocer el mes y el año.

Durante el cálculo de `CuantosDías` se hará una llamada a `EsBisiesto`, una función que determina el número de días de febrero.[2] Se usa también una función `Mannana` para el cálculo correcto del día siguiente, de forma que al domingo le siga el lunes.

El programa completo es el siguiente:

```
Program AlmanaqueSigloXX (input,output);
  {Calcula el día de la semana correspondiente a cada día del siglo XX}

  type
    tMeses = (ene, feb, mar, abr, may, jun, jul, ago, sep, oct,
              nov, dic);
    tDiasSemana = (lun, mat, mie, jue, vie, sab, dom);
    tAnnos = 1901..2000;
    tDiasMes = 1..31;
    tCalendarioSigloXX = array[tDıasMes, tMeses, tAnnos] of
                         tDıasSemana;

  var
    almanaque:  tCalendarioSigloXX;
    mes:  tMeses;
    ayer:  tDiasSemana;
    contAnnos:  tAnnos;
    dias, contaDias:  tDiasMes;
```

[2]En el desarrollo de esta función se tendrá en cuenta que los años múltiplos de 100 no son bisiestos, salvo que sean múltiplos de 400.

```
  function Mannana(hoy: tDiasSemana): tDiasSemana;
    {Dev. el día de la semana que sigue a hoy}
  begin
    if hoy = dom then
      Mannana:= lun  {se evita Succ(dom) ⇝ error}
    else
      Mannana:= Succ(hoy)
  end; {Mannana}

  function EsBisiesto(anno: tAnnos): boolean;
    {Efecto: Averigua si un año es bisiesto o no}
  begin
    EsBisiesto:= ((anno mod 4 = 0) and (anno mod 100 <> 0))
                 or (anno mod 400 = 0)
  end; {EsBisiesto}

  function CuantosDias(unmes: tMeses, anno: tAnnos): tDiasMes;
    {Dev. el numero de días del mes unmes del año anno}
  begin
    case unmes of
      abr,jun,sep,nov :
        CuantosDias:= 30;
      feb :
        if EsBisiesto(anno) then
          CuantosDias:= 29
        else
          CuantosDias:= 28;
      ene,mar,may,jul,ago,oct,dic :
        CuantosDias:= 31
    end {case}
  end; {CuantosDias}

begin {AlmanaqueSigloXX}
  {Crea un almanaque del siglo XX}
  ayer:= lun; {El dia 31 de diciembre de 1900 fue lunes}
  for contAnnos:= 1901 to 2000 do
    for mes:= ene to dic do begin
      dıas:= CuantosDias (mes, contAnnos);
        {número de días del mes actual}
      for contaDias:= 1 to dias do  begin
        almanaque[contaDias, mes, contAnnos]:= Mannana(ayer);
        ayer:= almanaque[contaDias,mes,contAnnos]
      end {for contaDias}
    end {for mes}
  end {for contAnnos}
end. {AlmanaqueSigloXX}
```

Para saber qué día de la semana fue el 12 de enero de 1925, se mirará la posición `almanaque[12,ene,1925]` (cuyo valor es `lun`).

12.4 Ejercicios

1. Complete adecuadamente los interrogantes de la definición que sigue,

```
const
  n = ?; {un entero positivo}
var
  i, j:  1..n;
  a :  array[1..n, 1..n] of ?
```

   e indique cuál es el efecto de la siguiente instrucción:

```
for i:= 1 to n do
  for j:= 1 to n do
    a[i, j]:= i=j
```

2. Considérese que los N primeros términos de una sucesión están registrados en un array de N componentes reales. Defina el subprograma `Sumar`, que transforma los elementos del array en los de la correspondiente serie:

$$a'_n \to \sum_{i=1}^{n} a_i$$

3. Escriba un programa que realice la suma de dos números positivos muy grandes (de 50 cifras, por ejemplo).

4. Para vectores de $\mathbb{R}^3$, defina subprogramas para calcular

   (a) el módulo: $\mathbb{R}^3 \to \mathbb{R}$

   (b) un vector unitario con su misma dirección

   (c) la suma: $\mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}^3$

   (d) el producto escalar: $\mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}$

   (e) el producto vectorial: $\mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}^3$

   (f) el producto mixto: $\mathbb{R}^3 \times \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}$

   Usando en lo posible los subprogramas anteriores, defina otros para averiguar lo siguiente:

   (a) dados dos vectores de $\mathbb{R}^3$, ver si uno de ellos es combinación lineal del otro.

   (b) dados tres vectores de $\mathbb{R}^3$, ver si uno de ellos es combinación lineal de los otros dos.

5. Escriba un subprograma que mezcle dos vectores (de longitudes m y n respectivamente), ordenados ascendentemente, produciendo un vector (de longitud $m+n$), también ordenado ascendentemente.

6. Escriba un subprograma que averigüe si dos vectores de N enteros son iguales. (La comparación deberá detenerse en cuanto se detecte alguna diferencia.)

7. Dados dos vectores de caracteres del mismo tipo, escriba un subprograma que averigüe si el primero de ellos precede al segundo en orden alfabético.

8. Escriba un subprograma que desplace todas las componentes de un vector de N enteros un lugar a la derecha, teniendo en cuenta que la última componente se ha de desplazar al primer lugar. Generalícese el subprograma anterior para desplazar las componentes k lugares.

9. Escriba un subprograma que lea una secuencia de caracteres del teclado registrándola en un vector de caracteres. Cuando el número de caracteres escritos sea inferior a la longitud del vector, se deberá rellenar con espacios en blanco por la derecha; y cuando haya más caracteres de los que caben, se eliminarán los últimos.

10. Si representamos la ecuación de una recta en el plano como un vector de tres componentes, $Ax + By + C = 0$, escriba subprogramas para determinar:

    (a) La ecuación de la recta que pasa por dos puntos dados.

    (b) La ecuación de la recta que pasa por un punto dado y tiene una cierta pendiente.

    (c) El ángulo que forman dos rectas dadas.

    (d) La distancia de un punto dado a una recta dada.

11. Defina una función que averigüe el máximo elemento de una matriz de $M \times N$ enteros.

12. Se tiene un sistema de ecuaciones lineales representado mediante una matriz de 3×4, donde las tres primeras columnas contienen los coeficientes del sistema (con determinante distinto de cero) y la cuarta los términos independientes. Escriba un subprograma para calcular la solución del sistema por la regla de Cramer.

13. Se tiene ahora un sistema de N ecuaciones con N incógnitas, supuestamente compatible determinado. Escriba un subprograma para calcular la solución del sistema por el método de Gauss.

14. Defina el tipo de una matriz cuadrada de dimensión N de elementos reales, y escriba un subprograma `Trasponer` que intercambie los elementos de posiciones (i, j) y (j, i) entre sí, $\forall i, j \in \{1, \ldots N\}$.

15. Defina un procedimiento para descomponer una matriz cuadrada M en otras dos, A y B, con sus mismas dimensiones, de manera que

$$M = S + A$$

    y tales que S es simétrica y A antisimétrica. Ello se consigue forzando que

$$S_{i,j} = S_{j,i} = \frac{M_{i,j} + M_{j,i}}{2}, \forall i, j \in \{1, \ldots N\}$$

y

$$A_{i,j} = -A_{j,i} = \frac{M_{i,j} - M_{j,i}}{2}, \forall i, j \in \{1, \ldots N\}$$

16. Defina subprogramas para determinar si una matriz es simétrica y si es triangular inferior, parando cuando se detecte que no lo es.

17. (a) Escriba un programa que lea los caracteres del `input` y efectúe una estadística de las letras que aparecen, contando la frecuencia de cada una de ellas sin tener en cuenta si es mayúscula o minúscula.

    (b) Modifique el programa del apartado (a) de manera que calcule cuántas veces aparece cada par de letras contiguas en el texto.

    (c) Modifique el programa del apartado anterior para que se muestre también cuántas veces aparece cada trío de letras contiguas en el texto.

18. Dada una matriz real de 3×3, escriba un programa que calcule:

    (a) Su determinante.

    (b) Su matriz adjunta.

    (c) Su matriz inversa.

19. (a) Escriba un subprograma para hallar el producto de dos matrices de $m \times n$ y de $n \times l$.

    (b) Escriba un subprograma que calcule la potencia de una matriz cuadrada de orden n. (Obsérvese que este problema admite un algoritmo iterativo y otro recursivo, como ocurre con la potencia de números.)

20. Un modo de averiguar el máximo elemento de un vector V de tamaño n es el siguiente: si el vector consta de un solo elemento, ése es el máximo; de lo contrario, se consideran los fragmentos $V_{1,\ldots,pm}$ y $V_{pm+1,\ldots,n}$ y se averigua el máximo en cada una de sus "mitades" (mediante este mismo procedimiento), resultando que el mayor de tales números es el de V.

    Desarrolle un subprograma que halle el máximo elemento de un vector en un fragmento dado mediante el procedimiento descrito.

21. Se tiene una frase en un array de caracteres. Desarrolle subprogramas para averiguar los siguientes resultados:

    (a) El número de palabras que contiene.

    (b) La longitud de la palabra más larga.

    (c) De todas las palabras, la que aparece antes en el diccionario.

    (Se entiende por palabra la secuencia de letras seguidas, delimitadas por un carácter que no es una letra o por los límites del array.)

Capítulo 13

Registros

En el capítulo anterior se ha estudiado cómo se trabaja en Pascal con colecciones de datos del mismo tipo. Pero en los problemas relacionados con la vida real es necesario agrupar con frecuencia datos de distintos tipos: por ejemplo, el documento nacional de identidad contiene, entre otros, un entero (el número) y cadenas de caracteres (el nombre, los apellidos, etc.); una ficha de artículo en un almacén debe contener datos numéricos (código, número de unidades en "stock", precio, etc.), booleanos (para indicar si se le aplica o no descuento) y cadenas de caracteres (la descripción del artículo, el nombre del proveedor, etc.). En este capítulo presentamos un nuevo tipo de datos, los registros, que nos van a permitir almacenar y procesar este tipo de información.

13.1 Descripción del tipo de datos registro

Los registros[1] son otro tipo de datos estructurados muy utilizados en Pascal. Su principal utilidad reside en que pueden almacenar datos de distintos tipos, a diferencia de los demás datos estructurados. Un registro estará formado por

[1]En inglés *record*.

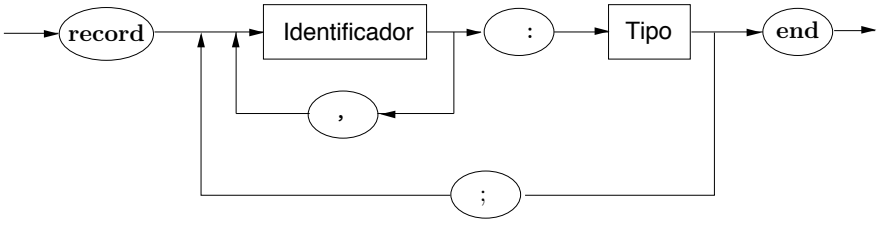


Figura 13.1.

varios datos (simples o estructurados) a los que llamaremos campos del registro y que tendrán asociado un identificador al que llamaremos nombre de campo.

La definición de un registro se hace según el diagrama sintáctico de la figura 13.1 y, por tanto, la definición de un registro genérico en Pascal es:

```
type
  tNombReg = record
    idenCampo1:  idTipo1;
    idenCampo2:  idTipo2;
    ...
    idenCampoN: idTipoN
  end; {tNombReg}
```

Por ejemplo, supongamos que un encargado de obra quiere tener registrados varios datos de sus trabajadores, tales como: nombre, dirección, edad y número de D.N.I. Con los tipos de datos que conocemos resultaría bastante difícil, ya que tendríamos que indicar que las variables `direccion`, `edad` y `dni` están relacionadas con el nombre de un trabajador en concreto. Para solucionarlo, se utiliza el tipo de datos estructurado *registro*, de la siguiente forma:

```
type
  tEdades = 16..65;
  tDigitos = '0'..'9';
  tFicha = record
    nombre:  array[1..30] of char;
    direccion:  array[1..50] of char;
    edad:  tEdades;
    dni:  array[1..8] of tDigitos
  end; {tFicha}
```

Como se puede observar, en este ejemplo se utilizan datos estructurados en la definición de otro dato estructurado (dentro de la estructura del dato *registro*

se utilizará un vector de caracteres para almacenar el `nombre` y la `direccion` de un empleado).

Es conveniente destacar que el tipo de datos registro, al igual que el tipo array, es un tipo estructurado de tamaño fijo; sin embargo se diferencian de ellos principalmente en que los componentes de un array son todos del mismo tipo, mientras que los componentes de un registro pueden ser de tipos distintos.

13.1.1 Manejo de registros: acceso a componentes y operaciones

El dominio de un registro estará formado por el producto cartesiano de los dominios de sus campos componentes.

Para poder trabajar con el tipo de datos registro es necesario saber cómo acceder a sus campos, cómo asignarles valores y que tipo de operaciones podemos realizar con ellos:

- Para acceder a los campos de los registros se utilizan construcciones de la forma `nomVarRegistro.nomCampo`, es decir, el nombre de una variable de tipo registro seguido de un punto y el nombre del campo al que se quiere acceder. Por ejemplo, si la variable `f` es de tipo `tFicha`, para acceder a sus campos `nombre, direccion, edad` y `dni` se utilizarán, respectivamente, las construcciones:

  ```
  f.nombre
  f.direccion
  f.edad
  f.dni
  ```

  En este punto se debe señalar que, a diferencia de los arrays, en los cuales el acceso se realiza por medio de índices (tantos como dimensiones tenga el array, que pueden ser el resultado de una expresión y por tanto calculables), en los registros se accede por medio de los identificadores de sus campos, que deben darse explícitamente.

- Los tipos de los campos pueden ser tipos predefinidos o definidos por el programador mediante una definición de tipo previa. Incluso un campo de un registro puede ser de tipo registro. Así, por ejemplo, si queremos almacenar para cada alumno, su nombre, fecha de nacimiento y nota, podríamos definir tipos y variables de la siguiente forma:

```
type
  tMeses  = (ene, feb, mar, abr, may, jun, jul, ago, sep, oct, nov, dic);
  tCalificaciones = (NP, Sus, Apr, Notab, Sob, MH);
  tNombre = array[1..50] of char;
  tFecha = record
    dia:  1..31;
    mes:  tMeses;
    anno:  1900..2000
  end; {tFecha}
  tFicha = record
    nombre:  tNombre;
    fechaNac:  tFecha;
    nota:  tCalificaciones
  end; {tFicha}
var
  alumno:  tFicha;
```

- La asignación de valores a los campos se hará dependiendo del tipo de cada uno de ellos. Así, en el ejemplo anterior, para iniciar los datos de la variable `Alumno` tendríamos que utilizar las siguientes asignaciones:

```
alumno.nombre:= 'Mario␣Aguilera␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣';
alumno.fechaNac.dia:= 3;
alumno.fechaNac.mes:= feb;
alumno.fechaNac.anno:= 1973;
alumno.nota:= Notab
```

- Las operaciones de lectura y escritura de registros han de hacerse campo por campo, empleando procedimientos o funciones especiales si el tipo del campo así lo requiere. Por ejemplo:

```
procedure EscribirFicha(unAlumno:  tFicha);
  {Efecto:  Escribe en la pantalla el contenido del registro
   unAlumno}
begin
  WriteLn('Nombre:  ', unAlumno.nombre);
  Write('Fecha de nacimiento:  ',unAlumno.fechaNac.dia);
  EscribirMes(unAlumno.fechaNac.mes);
  WriteLn(unAlumno.fechaNac.anno);
  Write('Nota:  ');
  EscribirNota(unAlumno.nota)
end; {EscribirFicha}
```

Obsérvese que los campos `fecha.mes` y `nota` son de tipo enumerado y necesitarán dos procedimientos especiales (`EscribirMes` y `EscribirNota`, respectivamente) para poder escribir sus valores por pantalla.

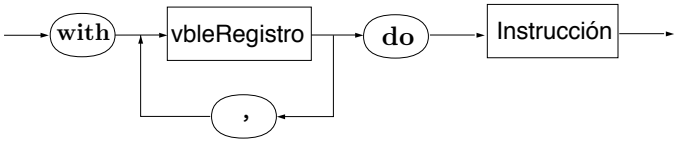


Figura 13.2.

Al igual que todos los tipos de datos compuestos, un registro no puede ser el resultado de una función. Para solucionar este problema actuaremos como de costumbre, transformando la función en un procedimiento con un parámetro por variable adicional de tipo registro que albergue el resultado de la función. Así, por ejemplo:

```
procedure LeerFicha(var unAlumno:  tFicha);
  {Efecto:  lee del input el contenido del registro unAlumno}
begin
  Write('Nombre del alumno:');
  LeerNombre(unAlumno.nombre);
  Write('Día de nacimiento:  ');
  ReadLn(unAlumno.fechaNac.dia);
  Write('Mes de nacimiento:  ');
  LeerMes(unAlumno.fechaNac.mes);
  Write('Año de nacimiento:  ');
  ReadLn(unAlumno.fechaNac.anno);
  Write('Calificación:  ');
  LeerNota(unAlumno.nota)
end; {LeerFicha}
```

- Como puede observarse, es incómodo estar constantemente repitiendo el identificador `unAlumno`. Para evitar esta repetición Pascal dispone de la instrucción **with** que se emplea siguiendo la estructura descrita en el diagrama sintáctico de la figura 13.2.

  Por ejemplo, para el procedimiento `LeerFicha` se podrían utilizar dos instrucciones **with** anidadas de la siguiente forma:

```
procedure LeerFicha(var unAlumno:  tFicha);
  {Efecto:  lee del input el contenido del registro unAlumno}
begin
  with unAlumno do begin
    WriteLn('Introduce el nombre del alumno:');
    LeerNombre(nombre);
    with fechaNac do begin
```

```
      Write('Día de nacimiento:  ');
      ReadLn(dia);
      Write('Mes de nacimiento:  ');
      LeerMes(mes);
      Write('Año de nacimiento:  ');
      ReadLn(anno)
    end; {with fechaNac}
    Write('Calificación:  ');
    LeerNota(nota)
  end {with unAlumno}
end; {LeerFicha}
```

- Al igual que los arrays, si dos variables `r1` y `r2` son del mismo tipo registro, se pueden realizar asignaciones de registros completos (con la instrucción `r1:= r2`) evitando así tener que ir copiando uno a uno todos los campos del registro.

13.1.2 Registros con variantes

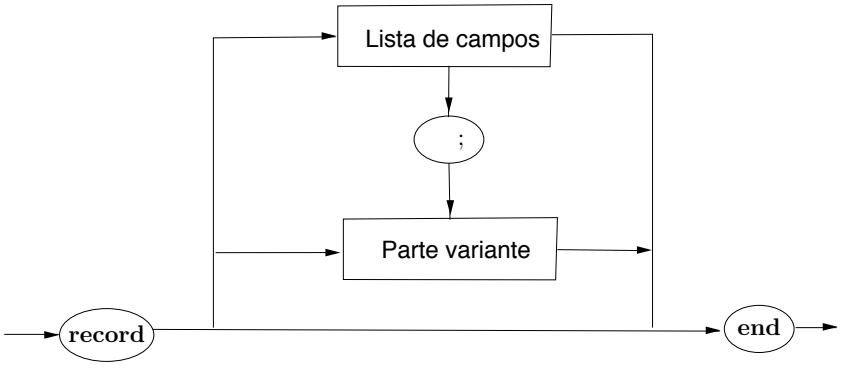
En ciertos casos es conveniente poder variar el tipo y nombre de algunos de los campos existentes en un registro en función del contenido de uno de ellos. Supongamos, por ejemplo, que en el registro `tFicha` definido anteriormente queremos incluir información adicional dependiendo de la nacionalidad. Si es española, añadiremos un campo con el D.N.I., y si no lo es, añadiremos un campo para el país de origen y otro para el número del pasaporte.

Con este objetivo se pueden definir en Pascal los registros con variantes, que constan de dos partes: la primera, llamada *parte fija*, está formada por aquellos campos del registro que forman parte de todos los ejemplares; la segunda parte, llamada *parte variable*, está formada por aquellos campos que sólo forman parte de algunos ejemplares.

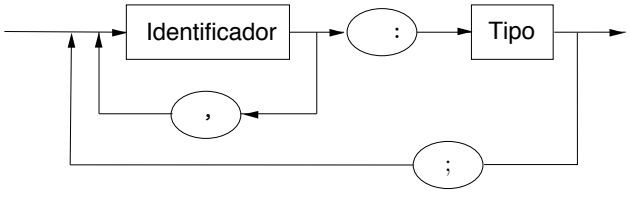
En la parte fija, debe existir un campo selector mediante el cual se determina la parte variable que se utilizará. Este campo selector debe ser único, es decir, sólo se permite un campo selector.

El diagrama sintáctico de la definición de un registro con variantes es el de la figura 13.3.

Para resolver el problema del ejemplo anterior se puede emplear un registro con variantes de la siguiente forma:



Lista de campos:



Parte variante:

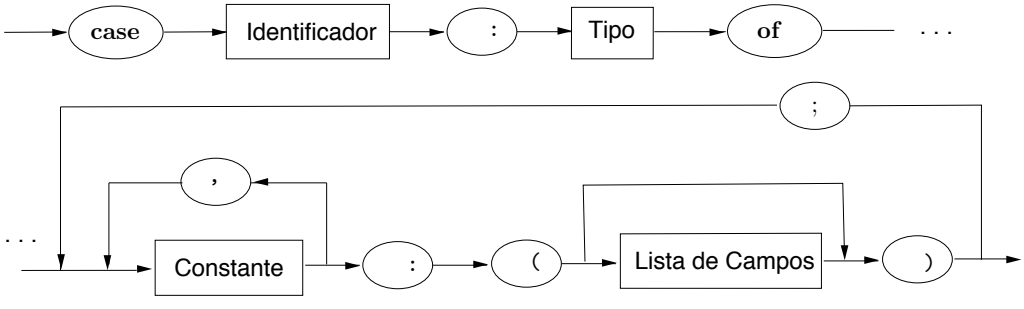


Figura 13.3.

```
type...
     tNombre = array[1..50] of char;
     tDNI = array[1..8] of '0'..'9';
     tPais = array[1..20] of char;
     tPasaporte = array[1..15] of '0'..'9';
     tFicha = record
       nombre: tNombre;
       fechaNac: tFecha;
       nota: tCalificaciones;
       case espannol : boolean of
         True :
           (dni : tDNI;
         False :
           (pais : tPais;
            pasaporte : tPasaporte
     end; {tFicha}
```

Con la definición anterior, el procedimiento `LeerFicha` queda como sigue:

```
procedure LeerFicha(var unAlumno: tFicha);
   {Efecto: lee del input el contenido del registro unAlumno}
   var
     c: char;
begin
   with unAlumno do begin
     Write('Introduce el nombre del alumno:');
     ReadLn(nombre);
     with fechaNac do begin
       Write('Día de nacimiento: ');
       ReadLn(dia);
       Write('Mes de nacimiento: ');
       LeerMes(mes);
       Write('Año de nacimiento: ');
       ReadLn(anno)
     end; {with fechaNac}
     Write('Calificacion: ');
     LeerNota(nota);
     repeat
       Write('¿Es ',nombre,' español? (S/N)');
       ReadLn(c)
     until c in ['s','S','n','N'];
     case c of
       's','S' :
         begin {el alumno es español}
           espannol:= True;
           Write('DNI: ');
```

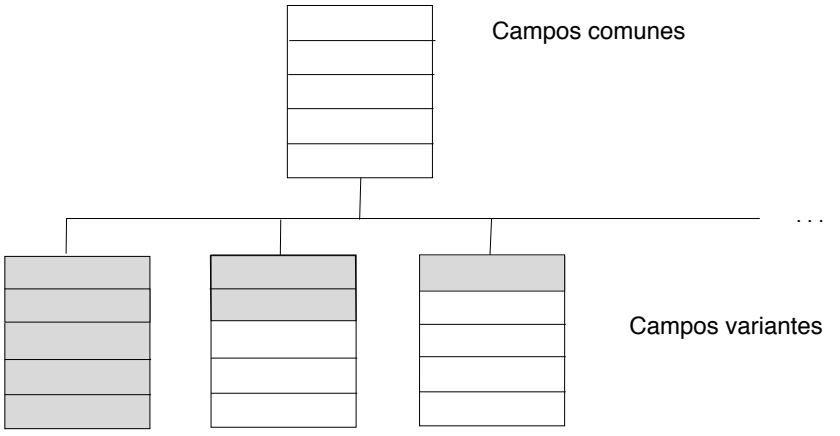


Figura 13.4.

```
          LeerDNI(dni)
        end;
      'n','N' :
        begin {el alumno es extranjero}
          espannol:= False;
          Write('país:  ');
          LeerPais(pais);
          Write('pasaporte:  ');
          LeerPasaporte(pasaporte)
        end
    end {case}
  end {with unAlumno}
end; {LeerFicha}
```

La utilización de registros con campos variantes relaja en parte la condición de tipos fuertes de Pascal al permitir que una variable de esta clase almacene valores de diferentes tipos. Sin embargo, debe tenerse cuidado con el uso de la parte variante, ya que los compiladores no suelen comprobar que esa parte se utiliza correctamente.

Al implementar los registros con variantes el compilador hace reserva de memoria para la variante más grande, aunque no se aproveche en el caso de las variantes más pequeñas, como se muestra en la figura 13.4.

13.2 Arrays de registros y registros de arrays

Dado que los registros permiten almacenar datos de diferentes tipos correspondientes a una persona u objeto y que los arrays agrupan datos de un mismo

tipo, es frecuente combinarlos formando arrays de registros que permitan almacenar y gestionar la información relativa a un grupo de personas u objetos.

Por ejemplo, una vez definido el tipo `tFicha` del apartado 13.1.1 donde almacenamos los datos de un alumno, podemos definir el tipo siguiente:

```
type
  tVectorFichas = array[1..40] of tFicha;
```

y declarar las variables:

```
var
  alumno:  tFicha;
  clase:  tVectorFichas;
```

De esta forma, en el vector `clase` podemos almacenar los datos de los alumnos de una clase. Para acceder al año de nacimiento del alumno número 3, tendríamos que utilizar la siguiente expresión:

```
clase[3].fecha.anno
```

mientras que para acceder a su nota usaríamos:

```
clase[3].nota
```

Los arrays tienen un tamaño fijo, sin embargo, hay casos en los que el número de datos no se conoce *a priori*, pero para los que puede suponerse un máximo.

En este caso, se puede definir un registro que contenga a un vector del tamaño máximo y una variable adicional para llevar la cuenta de la parte utilizada.

Por otra parte, hay casos en los que puede resultar útil definir registros de arrays. Supongamos, por ejemplo, que queremos calcular el valor medio de una variable estadística real de una muestra cuyo tamaño no supera los cien individuos. Para ello se podrían hacer las siguientes definiciones y declaraciones:

```
const
  MaxCompo = 100;
type
  tVector = array [1..MaxCompo] of real;
  tRegistro = record
    vector:  tVector;
    ocupado:  0..MaxCompo
  endtRegistro;
var
  indice:  1..MaxCompo;
  reg:  tRegistro;
  valor, suma, media:  real;
  fin:  boolean;
```

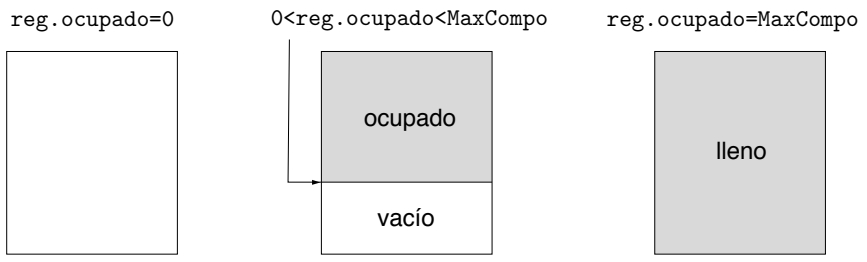


Figura 13.5.

Para introducir los datos hay que comprobar si el vector está lleno y, en caso contrario, se pide el dato y se incrementa el valor de `reg.ocupado`, que almacena el índice del último elemento introducido. Si `reg.ocupado` vale cero, el vector está vacío, si $0 <$ `reg.ocupado` $<$ `MaxCompo` tiene `reg.ocupado` datos y quedan `MaxCompo` $-$ `reg.ocupado` espacios libres, y por último, cuando `reg.ocupado` $=$ `MaxCompo`, el vector está lleno. Todas estas situaciones se recogen en la figura 13.5.

Para introducir los valores en el vector podríamos escribir un fragmento de programa como el siguiente:

```
reg.ocupado:= 0;
fin:= False;
while (reg.ocupado < MaxCompo) and not(fin) do begin
    Write('Introduzca el valor del individuo ', reg.ocupado + 1)
    Write(' o un valor negativo para terminar ');
    ReadLn(valor);
    if valor >= 0 then begin
      reg.ocupado:= reg.ocupado + 1;
      reg.vector[reg.ocupado]:= valor
    end
    else
      fin:= True
end {while}
```

Para calcular la media se recorre la parte ocupada del vector acumulando sus valores, como se muestra a continuación:

```
for indice:= 1 to reg.ocupado do
  suma:= suma + reg.vector[indice];
media:= suma/reg.ocupado
```

13.3 Ejercicios

1. Defina un tipo de datos para manejar fechas, incluyendo la información usual para un día cualquiera del calendario: el número de día dentro del mes, el día de la semana, el mes y el año, y con él, los siguientes subprogramas:
   (a) Lectura y escritura.
   (b) Avance (que pasa de un día al siguiente), con ayuda de una función que indica el número de días de un mes de un cierto año.
   (c) Distancia entre fechas, usando la función avance.
2. Considérese una representación de los números reales mediante su signo, positivo o negativo, su parte entera, formada por `N` dígitos (por ejemplo, 25) de cero a nueve y por su parte decimal, formada por `NDEC` cifras (por ejemplo, 5).
   (a) Defina en Pascal este tipo de datos.
   (b) Defina procedimientos apropiados para su lectura y escritura.
   (c) Defina un subprograma para sumar reales, controlando el posible desbordamiento.
   (d) Defina un subprograma para comparar reales.
3. (a) Defina un tipo de datos registro que permita almacenar un punto del plano real y un tipo vector formado por tres registros del tipo anterior.[2]
   (b) Escriba un subprograma que determine si los tres puntos almacenados en una variable del tipo vector forman un triángulo.
   (c) Escriba un subprograma tal que, si tres puntos forman un triángulo, calcule su área aplicando la fórmula de Herón (véase el ejercicio 6 del capítulo 4).
4. (a) Defina un tipo registro que permita almacenar un ángulo dado en forma de grados (sexagesimales), minutos y segundos.
   (b) Escriba dos subprogramas, el primero para convertir un ángulo dado en radianes en una variable del tipo registro anterior, y el segundo para realizar la conversión inversa. En el primero se tendrá en cuenta que el resultado debe tener el número de grados inferior a 360°, el de minutos inferior a 60' y el de segundos inferior a 60".
5. La posición de un objeto lanzado con velocidad inicial v y con ángulo α con respecto a la horizontal, transcurrido un cierto tiempo t, puede expresarse (despreciando el rozamiento) con las siguientes ecuaciones:

$$x = vt\cos\alpha \qquad y = vt\,\mathrm{sen}\,\alpha - \frac{1}{2}gt^2$$

   donde g es la aceleración de la gravedad ($9.8m/seg^2$).
   (a) Defina un registro que almacene la velocidad inicial y el ángulo α.

[2]Obsérvese que ésta es una definición alternativa a la de arrays, tal vez más apropiada por ser las componentes del mismo tipo.

(b) Defina un registro que almacene las coordenadas x, y y el tiempo t transcurrido desde el lanzamiento.

(c) Escriba un subprograma que utilice un registro de cada uno de los tipos anteriores y calcule la posición del objeto a partir de v, α y t.

(d) Escriba un subprograma que calcule la altura máxima aproximada alcanzada por el objeto utilizando intervalos de tiempo "pequeños", por ejemplo décimas o centésimas de segundo.

(e) Escriba un subprograma que calcule la distancia máxima aproximada alcanzada por el objeto, con la misma técnica del apartado anterior. Indicación: la distancia máxima se alcanza cuando la altura se hace cero.

(f) Redefina los registros de los apartados anteriores para utilizar ángulos en forma de grados (sexagesimales), minutos y segundos, realizando las conversiones con los subprogramas del apartado 4b.

6. Un comercio de alimentación almacena los siguientes datos de sus productos: producto (nombre del producto), marca (nombre del fabricante), tamaño (un número que indica el peso, volumen, etc. de un determinado envase del producto), precio (del tamaño correspondiente) y unidades (cantidad existente en inventario).

(a) Defina un tipo de datos registro que permita almacenar dichos campos.

(b) Defina un vector suficientemente grande de dichos registros.

(c) Defina un registro que incluya el vector y un campo adicional para indicar la parte ocupada del vector de acuerdo con la técnica expuesta en el apartado 13.2.

(d) Escriba los subprogramas necesarios para realizar las siguientes operaciones:

- **Altas:** Consiste en introducir nuevos productos con todos sus datos. Normalmente se efectuarán varias altas consecutivas, deteniendo el proceso cuando se introduzca un producto cuyo nombre comience por una clave especial (un asterisco, por ejemplo).

- **Bajas:** Se introduce el nombre del producto a eliminar, se muestran sus datos y se pide al usuario confirmación de la baja. Para eliminar el producto se desplazan desde el producto inmediato siguiente una posición hacia delante los sucesivos productos, y se reduce en una unidad la variable que indica la posición ocupada. Indicación: se deben tratar de forma especial los casos en que no hay productos, cuando sólo hay uno y cuando se elimina el último.

- **Modificaciones:** Se introduce el nombre del producto, se muestra y se piden todos sus datos.

- **Consultas:** Pueden ser de dos tipos: por producto, pidiendo el nombre y mostrando todos sus datos, y total, mostrando el listado de todos los productos con sus datos. Se puede añadir el coste total por producto y el coste total del inventario. Se puede mostrar por pantalla y por impresora.

(e) Escriba un programa completo comandado por un menú con las operaciones del apartado anterior.

7. Construya un tipo registro para manejar una hora del día, dada en la forma de horas (entre 0 y 23), minutos y segundos (entre 0 y 59). Utilizando este tipo, escriba subprogramas para:

   (a) Leer correctamente un instante dado.

   (b) Mostrar correctamente una hora del día.

   (c) Dado un tiempo del día, pasarlo a segundos.

   (d) Dados dos tiempos del día, calcular su diferencia, en horas, minutos y segundos.

   (e) Dado un tiempo del día, calcular el instante correspondiente a un segundo después.

   (f) Dado un tiempo del día, mostrar un reloj digital en la pantalla durante un número de segundos predeterminado. (El retardo se puede ajustar con bucles vacíos.)

8. La posición de un punto sobre la superficie de la tierra se expresa en función de su longitud y latitud. La primera mide el ángulo que forma el meridiano que pasa por el punto con el meridiano que pasa por el observatorio de Greenwich, y toma valores angulares comprendidos entre 0 y 180^o, considerándose longitud Este (E) cuando el ángulo se mide hacia el Este y longitud Oeste (W) en caso contrario. La segunda mide el ángulo que forma la línea que pasa por el punto y por el centro de la tierra con el plano que contiene al ecuador, y toma valores angulares comprendidos entre 0 y 90^o, considerándose latitud Norte (N) cuando el punto está situado al Norte del ecuador y latitud Sur (S) en caso contrario. En ambos casos los valores angulares se miden en grados, minutos y segundos. Defina un tipo registro que permita almacenar los datos anteriores. Escriba subprogramas para leer y escribir correctamente variables del tipo anterior, en grados, minutos y segundos.

9. Escriba dos tipos registro, uno para almacenar las coordenadas cartesianas de un punto del plano y otro para almacenarlo en coordenadas polares con el ángulo en radianes. Escriba subprogramas para pasar de unas coordenadas a otras. Escriba un subprograma que calcule la distancia entre dos puntos en coordenadas cartesianas, polares y ambas.

10. Dada una lista de puntos del plano, en un vector no completo (véase el apartado 13.2) que se supone que definen los vértices de un polígono, determine el área del mismo mediante la fórmula siguiente:

$$A = \frac{1}{2}((X_2Y_1 - X_1Y_2) + (X_3Y_2 - X_2Y_3) + (X_4Y_3 - X_3Y_4) + \ldots + (X_1Y_N - X_NY_1))$$

Capítulo 14

Archivos

Como se dijo en [PAO94] (véanse los apartados 4.2.2 y 6.1.1) los archivos se pueden entender como un conjunto de datos que se almacenan en un dispositivo de almacenamiento, por ejemplo, una unidad de disco.

Para expresar este concepto, Pascal dispone del tipo de datos estructurado **file** (*archivo* en inglés), que se explica en los apartados siguientes.

14.1 Descripción del tipo de datos archivo

Un archivo en Pascal se estructura como una *secuencia* homogénea de datos, de tamaño no fijado de antemano, la cual se puede representar como una fila de celdas en las que se almacenan los datos componentes del archivo. Una marca especial llamada *fin de archivo* señala el fin de la secuencia. La estructura del archivo se muestra gráficamente en la figura 14.1.

De hecho, hemos trabajado con archivos en Pascal desde el principio de este texto, ya que, como sabemos, los programas que producen alguna salida por pantalla necesitan el archivo estándar `output`, que se incluye en el encabezamiento

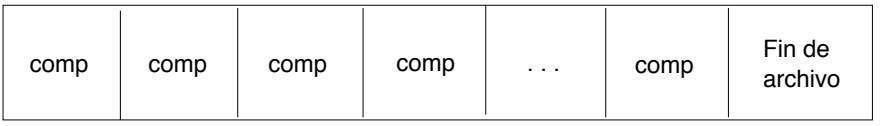


Figura 14.1.

de la siguiente forma:[1]

```
Program NombrePrograma (output);
```

Si el programa requiere, además, que se introduzca algún valor por teclado, necesitaremos también el archivo estándar `input` que debe entonces declararse en el encabezamiento del programa:

```
Program NombrePrograma (input, output);
```

En este apartado se detalla cómo utilizar otros archivos que sirvan para la lectura y escritura de datos. Estos nuevos tipos de archivos de entrada y salida se asociarán a archivos almacenados en unidades de disco. Pascal permite acceder a ellos para guardar datos que posteriormente podrán ser leídos por el mismo o por otro programa.

Imaginemos que se ejecuta el programa `AlmanaqueSigloXX` que aparece como ejemplo en el apartado 12.3. Una vez que hemos ejecutado dicho programa sería de gran utilidad almacenar los resultados obtenidos en un archivo y, así, poder utilizarlos posteriormente sin tener que generar nuevamente el almanaque, con el consiguiente ahorro de tiempo. En los siguientes apartados se presentan los archivos de Pascal en general, y se detalla el caso particular de los archivos de texto, por su frecuente utilización.

Los archivos tienen como limitación el que sus elementos no pueden ser archivos. Por lo tanto, no es posible la declaración

```
tArchivo = file of file of ...;
```

14.2 Manejo de archivos en Pascal

Un archivo es un tipo de datos estructurado que permitirá almacenar en una unidad de disco información homogénea, es decir, datos de un mismo tipo, ya sea

[1]Es conveniente incluir siempre en el encabezamiento del programa el archivo de salida `output` aunque no se vaya a realizar ninguna salida por pantalla, ya que ésto permitirá al programa escribir en pantalla los mensajes de error que pudieran originarse.

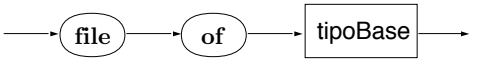


Figura 14.2.

básico o estructurado, por lo que las componentes del archivo van a ser valores de este tipo.

Como es natural, antes de trabajar con el tipo de datos archivo es necesario conocer su definición. El diagrama sintáctico de la definición de archivos es el de la figura 14.2.

Por ejemplo, si queremos trabajar con un archivo cuyos elementos sean arrays de caracteres, utilizaremos la siguiente definición:

```
type
  tTarjeta = array[1..50] of char;
  tArchivo = file of tTarjeta;
var
  unaTarjeta:  tTarjeta;
  archivoTarjetas:  tArchivo;
```

Si en un programa Pascal se va a utilizar un archivo externo, es necesario incluir el identificador del archivo (por ejemplo `nombreArchivo`) en el encabezamiento. Por tanto, un programa que maneje el archivo `nombreArchivo`, debe ser declarado como sigue:

```
Program TratamientoDeArchivo (input, output, nombreArchivo);
```

Con esta declaración el programa ya reconocerá al archivo con el que vamos a trabajar y estará preparado para poder realizar operaciones de acceso a dicho archivo.[2] Así, en el ejemplo anterior debemos realizar la siguiente declaración de programa:

```
Program TratamientoDeTarjetas (input, output, archivoTarjetas);
```

[2]Además, es necesario asociar el archivo "lógico" (declarado en el programa) con un archivo "físico" en el disco (véase el apartado B.9). Para realizar esta operación en Turbo Pascal se utiliza la instrucción `Assign (nombreArchivoLogico, NombreArchivoFisico)`. Por ejemplo, la instrucción

`Assign (archivoTarjetas, 'C:\TARJETAS\DATOS.TXT')`

indica que los datos de `archivoTarjetas` se almacenarán en el archivo de disco `DATOS.TXT` dentro del subdirectorio `TARJETAS` de la unidad `C:`.

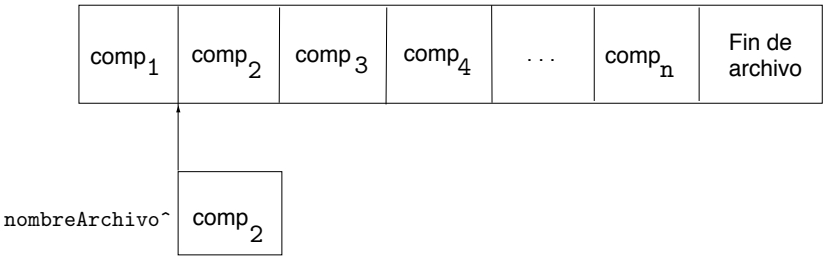


Figura 14.3.

Para acceder a las componentes de un archivo se utiliza el llamado *cursor* del archivo. Este cursor se puede entender como una "ventana" por la cual vemos una componente del archivo (aquélla a la que apunta), como se muestra en la figura 14.3.

La notación utilizada en Pascal para referenciar el cursor de un archivo es:[3]

```
nombreArchivo^
```

Como se citó al comienzo de este capítulo, todo archivo tiene asociada una marca de fin de archivo. En Pascal se dispone de una función booleana llamada `EoF`.[4] Para utilizarla tendremos que indicarle el nombre del archivo. Así, `EoF(nombreArchivo)` devuelve el valor `False` si no se ha alcanzado el final del archivo o `True` en caso contrario (en cuyo caso el contenido del cursor `nombreArchivo^` es indeterminado).

Hemos de destacar en este momento que, como se puede comprobar, el manejo de archivos en Pascal no es muy eficiente debido a que sólo se dispone de archivos secuenciales.

14.2.1 Operaciones con archivos

Las operaciones más importantes que se pueden realizar con los archivos son la escritura y lectura de sus componentes. Estas operaciones se van a llevar a cabo de forma muy similar a la lectura y escritura usual (con las instrucciones `Read` y `Write`), salvo que se redireccionarán al archivo adecuado.

Un archivo se crea o se amplía escribiendo en él. Cada vez que se realice una operación de escritura, se añadirá una nueva componente al final del archivo

[3]En Turbo Pascal no se puede utilizar directamente el cursor `nombreArchivo^` ni las operaciones `Put` y `Get` que se presentan más adelante; posiblemente ésta es la razón por la que esta notación ha caído en desuso. (Véase el apartado B.9 para ver las operaciones equivalentes.)

[4]Del inglés *End Of File* (fin de archivo).

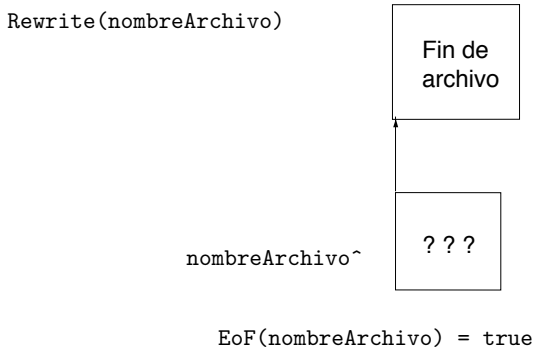


Figura 14.4.

secuencial. El cursor del archivo avanzará una posición cada vez que se escriba o se lea en el archivo.

La creación de un archivo se hace mediante la siguiente instrucción:

```
Rewrite(nombreArchivo)
```

que sitúa el cursor al principio del archivo `nombreArchivo` y, además, destruye cualquier posible información existente en él, como se muestra en la figura 14.4. Una vez ejecutada la instrucción anterior, el archivo está preparado para recibir operaciones de escritura como la siguiente:

```
Put(nombreArchivo)
```

que añade una componente más al archivo en la posición que indique el cursor y avanza éste un lugar en el archivo `nombreArchivo`, como puede verse en la representación gráfica de la figura 14.5. Después de ejecutar dicha instrucción, `nombreArchivo^` queda indefinido.

Así, teniendo en cuenta la declaración hecha anteriormente y suponiendo que la variable `unaTarjeta` posee la información que queremos almacenar en el archivo `archivoTarjetas`, realizaremos las siguientes instrucciones para añadir una nueva componente al archivo:

```
archivoTarjetas^:= unaTarjeta;
Put(archivoTarjetas)
```

Estas dos instrucciones son equivalentes a la instrucción:

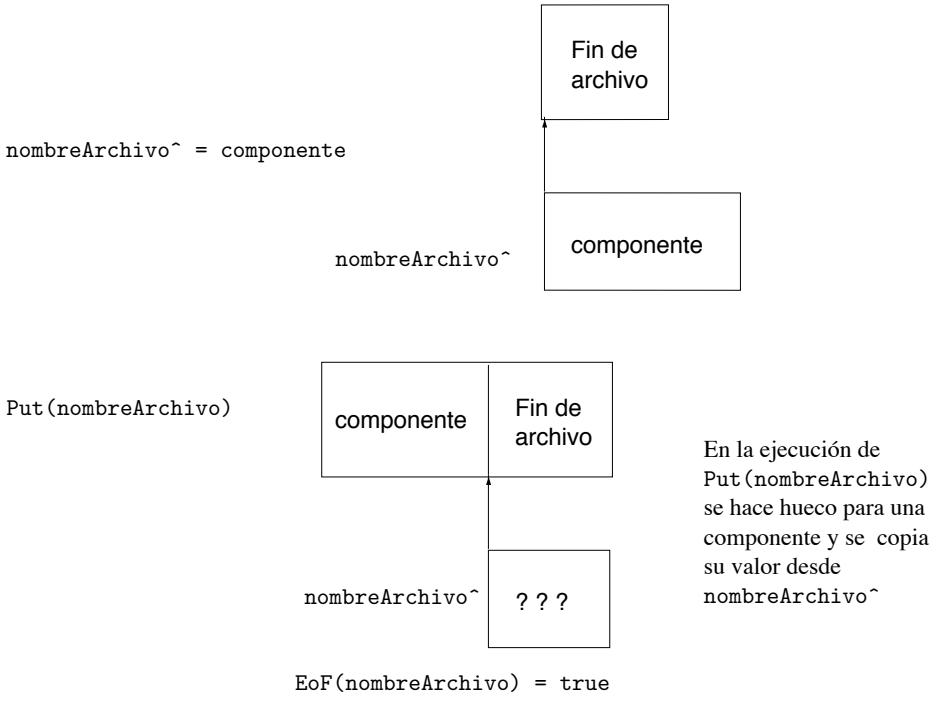

Fin de archivo
nombreArchivo^ = componente
componente
nombreArchivo^
Put(nombreArchivo)
componente
Fin de archivo
En la ejecución de Put(nombreArchivo) se hace hueco para una componente y se copia su valor desde nombreArchivo^
nombreArchivo^
? ? ?
EoF(nombreArchivo) = true

Figura 14.5.

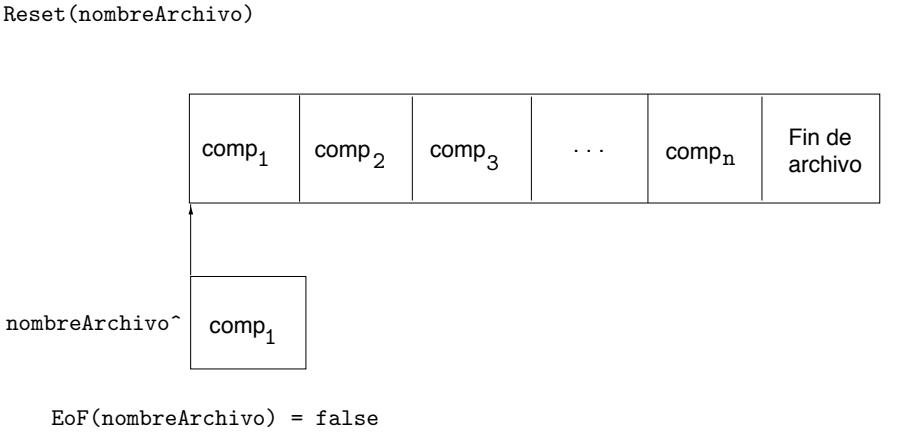


Figura 14.6.

```
Write(archivoTarjetas, unaTarjeta);
```

Dado que Pascal trabaja con archivos secuenciales, el cursor está siempre situado al final del archivo cuando se va a realizar una operación de escritura. Las sucesivas componentes se van añadiendo por el final del archivo, desplazando la marca de fin de archivo.

Una vez que hemos creado un archivo, es importante poder leer sus componentes. Dado que el acceso se hace de forma secuencial, hay que situar, nuevamente, el cursor al principio del archivo. Para ello, se ejecutará la instrucción:

```
Reset(nombreArchivo)
```

Con esta instrucción también se coloca el cursor en la primera componente del archivo. Si el archivo no está vacío, su primera componente está disponible en la variable `nombreArchivo^`, como puede comprobarse en la figura 14.6.


Obsérvese que las funciones `Rewrite` y `Reset` son muy parecidas, ya que ambas sitúan el cursor al principio del archivo. La diferencia existente entre ambas es que la primera prepara el archivo exclusivamente para escritura (destruyendo la información existente), mientras que la segunda lo prepara exclusivamente para lectura.

Una vez que el cursor apunte a la primera componente, se puede mover el cursor a la siguiente posición y copiar la información de la siguiente componente de `nombreArchivo` utilizando la instrucción

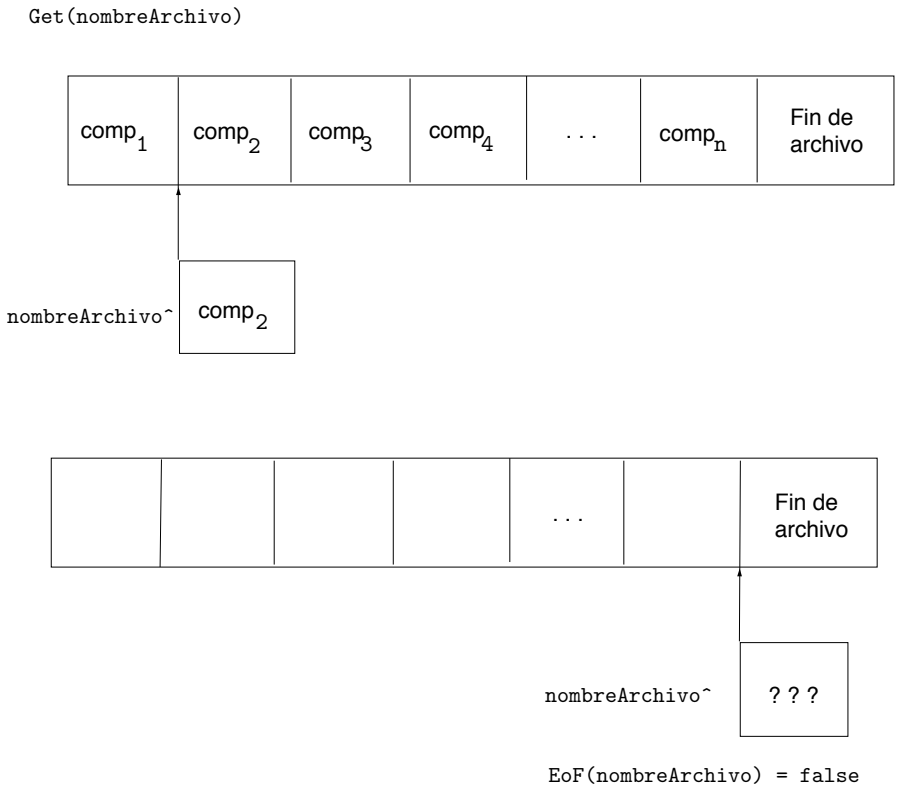


Figura 14.7.

```
Get(nombreArchivo)
```

Su efecto se muestra en la figura 14.7.

Siguiendo con el ejemplo anterior, si deseamos leer el contenido de la componente del archivo apuntada por el cursor realizaremos las siguientes instrucciones:

```
unaTarjeta:= archivoTarjetas^;
Get(archivoTarjetas)
```

o, equivalentemente,

```
Read(archivoTarjetas, unaTarjeta)
```

Es muy importante tener en cuenta que antes de leer de un archivo se ha de comprobar, mediante la función `EoF`, que quedan componentes por leer. Esta

función es imprescindible para realizar la lectura de archivos en los que desconocemos a priori el número de componentes o para detectar archivos vacíos.

En el siguiente ejemplo se presenta el esquema de un programa que lee un archivo completo controlando el final de archivo con `EoF`:

```
Program LeerArchivo (input, output, archivoTarjetas);

  type
    tTarjeta = array[1..50] of char;
    tArchivo = file of tTarjeta;
  var
    archivoTarjetas:  tArchivo;
    una Tarjeta:  tTarjeta
begin {LeerArchivo}
  ...
  Reset(archivoTarjetas);
  while not EoF(archivoTarjetas) do begin
    Read(archivoTarjetas, unaTarjeta);
    Procesar unaTarjeta
  end; {while}
end. {LeerArchivo}
```

El principal inconveniente que presentan los archivos en Pascal es que no se pueden alternar las operaciones de lectura y escritura en un archivo. Por tanto, si deseamos escribir y leer un archivo, en primer lugar se tendrá que escribir en él, y posteriormente situar el cursor al principio del archivo para leerlo.

Para realizar una copia de un archivo no se puede utilizar la asignación, a diferencia de los demás tipos de datos. Para poder copiar un archivo en otro se debe desarrollar un procedimiento cuyo código podría ser:

```
type
  tTarjeta = array[1..50] of char;  {por ejemplo}
  tArchivoTarjetas = file of tTarjeta;
...
procedure CopiarArchivo(var archiEnt, archiSal:  tArchivoTarjetas);
  {Efecto:  archiSal:= archiEnt}
  var
    unaTarjeta:  tTarjeta;
begin
  Reset(archiEnt);
  Rewrite(archiSal);
  while not EoF(archiEnt) do begin
    Read(archiEnt, unaTarjeta);
    Write(archiSal, unaTarjeta)
  end {while}
end; {CopiarArchivo}
```

14.3 Archivos de texto

Son muy frecuentes los programas en los que es necesario manejar textos, entendidos como secuencias de caracteres de una longitud usualmente grande, como, por ejemplo, una carta, un formulario, un informe o el código de un programa en un lenguaje de programación cualquiera. Estos textos se almacenan en archivos de caracteres que reciben el nombre de archivos de texto.

Los archivos de texto en Pascal se definen utilizando el tipo predefinido `text`. Este tipo de datos es un archivo con tipo base `char` al que se añade una marca de fin de línea. La generación y tratamiento del fin de línea se realiza con los procedimientos `WriteLn(archivoDeTexto)` y `ReadLn(archivoDeTexto)` y la función `EoLn(archivoDeTexto)`, que no se pueden utilizar con el tipo **file**.

El tipo `text` es un tipo estándar predefinido en Pascal, como `integer` o `char`, y por lo tanto, se pueden declarar variables de este tipo de la siguiente forma:

```
var
  archivoDeTexto:  text;
```

En particular, los archivos `input` y `output` son de texto y representan la entrada y salida estándar, que en vez de emplear un disco utilizan, normalmente, el teclado y la pantalla como origen y destino de las secuencias de caracteres. Siempre que se utiliza una instrucción `Read` o `Write` sin indicar el archivo, se asume que dichas operaciones se realizan sobre los archivos `input` y `output`, respectivamente.

Ambos archivos, como es sabido, deben ser incluidos en el encabezamiento de todo programa sin redeclararlos dentro del programa, ya que se consideran declarados implícitamente como:

```
var
  input, output :  text;
```

Además de esta declaración implícita se asumen, para los archivos `input` y `output`, las instrucciones `Reset(input)` y `ReWrite(output)` efectuadas al empezar un programa.

Debido a que los archivos de texto son muy utilizados, Pascal proporciona, además de las instrucciones comunes a todos los archivos con tipo genérico, funciones específicas de gran utilidad para los archivos de texto. Si `archivoDeTexto` es una variable de tipo `text`, según lo que vimos en el apartado anterior, sólo podríamos realizar instrucciones de la forma `Read(archivoDeTexto, c)` o bien `Write(archivoDeTexto, c)` siempre que `c` fuese de tipo `char`. Sin embargo, para los archivos `input` y `output` se permite que `c` pueda ser también de tipo `integer, real` o incluso `boolean`[5] o un array de caracteres para el caso de

[5]Sólo para archivos de salida.

`Write`.[6] Pascal permite este hecho no sólo a los archivos `input` y `output`, sino a todos los archivos de tipo `text`.

Pascal también permite que las instrucciones `Read` y `Write` tengan varios parámetros cuando se usan archivos de texto. Así, la instrucción

```
Read(archivoDeTexto, v1, v2, ..., vN)
```

es equivalente a:

```
Read(archivoDeTexto, v1);
Read(archivoDeTexto, v2);
...
Read(archivoDeTexto, vN)
```

y la instrucción

```
Write(archivoDeTexto, e1, e2, ..., eM)
```

es equivalente a:

```
Write(archivoDeTexto, e1);
Write(archivoDeTexto, e2);
...
Write(archivoDeTexto, eM)
```

donde `archivoDeTexto` es un archivo de tipo `text`, los parámetros `v1, v2,..., vN` pueden ser de tipo `integer, real` o `char` y los parámetros `e1,e2,...,eM` pueden ser de tipo `integer, real, char, boolean` o un array de caracteres. Las operaciones de lectura y escritura en archivos de texto son similares a los de `input` o `output` (véanse los apartados 4.3.2 y 4.3.3, donde se detalla su funcionamiento).

Los archivos de texto pueden estructurarse por líneas. Para manejar este tipo de organización existen la función `EoLn(archivoDeTexto)` para detectar el fin de línea y las instrucciones de lectura y escritura `ReadLn(archivoDeTexto)` y `WriteLn(archivoDeTexto)`. Su funcionamiento se describe a continuación:

- La función booleana `EoLn`[7] devuelve el valor `True` si se ha alcanzado la marca de fin de línea o `False` en otro caso. Cuando `EoLn(ArchivoDeTexto)` $\rightsquigarrow$ `True` el valor de la variable apuntado por el cursor (`archivoDeTexto^`)

[6] En estos casos se ejecutarán automáticamente subprogramas de conversión.

[7] Del inglés *End Of LiNe* (fin de línea.)

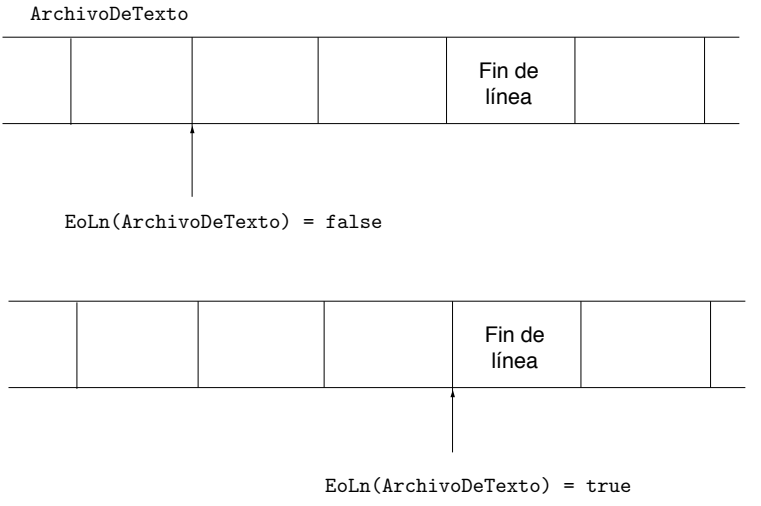


Figura 14.8.

es un caracter especial de fin de línea.[8] En la figura 14.8 puede verse una representación gráfica.

- La instrucción `ReadLn(archivoDeTexto, v)` lee el siguiente elemento del archivo, lo almacena en `v` y salta todos los caracteres hasta llegar al carácter especial de fin de línea, es decir, la instrucción es equivalente a:

```
Read(archivoDeTexto, v);
while not EoLn(archivoDeTexto) do
  Get(archivoDeTexto);
  {se avanza el cursor hasta encontrar el fin de línea}
Get(archivoDeTexto)
  {se salta el fin de línea}
```

  con lo que con la siguiente llamada a `Read` se leerá el primer carácter de la siguiente línea. El parámetro `v` es opcional. Si se omite, el efecto de la instrucción `Read(archivoDeTexto)` sería el mismo salvo que no se almacena la componente que esté actualmente apuntada por el cursor.

  De la misma forma que `Read`, la instrucción `ReadLn` puede utilizarse con el formato:

[8] En realidad este carácter especial depende del compilador que se utilice. Así, por ejemplo, en Pascal estándar se devuelve un espacio en blanco, mientras que Turbo Pascal devuelve el carácter de alimentación de línea (*Line Feed*, número 10 del juego de caracteres ASCII) seguido de un retorno de carro (*Carriage Return*, número 13 del juego de caracteres ASCII).

```
ReadLn(archivoDeTexto, v1, v2, ..., vN)
```

que es equivalente a:

```
Read(archivoDeTexto, v1);
Read(archivoDeTexto, v2);
...
ReadLn(archivoDeTexto, vN)
```

- La instrucción `WriteLn(archivoDeTexto, expresión)` se usa para escribir el valor de `expresión` en el `archivoDeTexto` y poner la marca de fin de línea. Asimismo, podemos omitir la `expresión`, produciéndose entonces simplemente un salto de línea.

  Existe también la posibilidad de utilizar la instrucción:

```
WriteLn(archivoDeTexto, e1, e2, ..., eM)
```

que es equivalente a:

```
Write(archivoDeTexto, e1);
Write(archivoDeTexto, e2);
...
WriteLn(archivoDeTexto, eM)
```

Como una generalización de lo dicho en este tema sobre el uso de las funciones `EoLn` y `EoF`, se observa que la estructura general de los programas que procesan archivos de texto constan frecuentemente de dos bucles anidados: uno controlado por `EoF` y otro por `EoLn`, como se muestra a continuación, en un programa genérico de procesamiento de archivos de texto.

```
Program LecturaArchivoTexto (input, output, archivoDeTexto);
  Definición de tipos y declaración de variables
begin {Proceso de archivo de texto}
  ...
  Reset(archivoDeTexto);
  while not EoF(archivoDeTexto) do begin
    while not EoLn(archivoDeTexto) do begin
      Read(archivoDeTexto, dato);
```

```
        Procesar dato
      end; {while not EoLn}
      ReadLn(archivoDeTexto)
    end {while not EoF}
    ...
  end.  {LecturaArchivoTexto}
```

En cualquiera de los casos considerados anteriormente, si se omite el archivo `archivoDeTexto` se supondrá por defecto el archivo `input` para el caso de `Read` o el archivo `output` para el caso de `Write`, produciéndose, en tal caso, una lectura por teclado o una escritura por pantalla respectivamente.

Finalmente se debe destacar que los archivos, al igual que todo tipo de datos compuesto, se pueden pasar como parámetros en funciones y procedimientos, pero no pueden ser el resultado de una función. No obstante, siempre se deben pasar como parámetros por variable.

14.4 Ejercicios

1. Basándose en el ejercicio 2 del capítulo 10 desarrolle un programa para las siguientes tareas:

   (a) Invertir una serie de líneas, manteniendo su orden.

   (b) Copiar una serie de líneas en orden inverso.

2. Escriba un programa que tabule los coeficientes binomiales $\binom{n}{k}$ (véase el ejercicio 10 del capítulo 6), confeccionando el archivo `BINOM.TXT` de la siguiente forma:

   ```
   1
   1␣1
   1␣2␣1
   1␣3␣3␣1
   1␣4␣6␣4␣1
   1␣5␣10␣10␣5␣1
   ...
   ```

   Escriba una función que, en vez de hallar el coeficiente $\binom{n}{k}$, lo consulte en la tabla creada.

3. Se desea tabular los valores de la función de distribución normal

$$f(t) = \frac{1}{2} + \frac{1}{\sqrt{2\pi}} \int_0^t e^{-x^2/2} dx$$

   para los valores de t entre $0'00$ y $1'99$, aumentando a pasos de una centésima (véase el ejercicio 11 del capítulo 6).

Desarrolle un programa que construya un archivo de texto `NORMAL.TXT` con veinte filas de diez valores, así:

$$\begin{array}{cccc} f(0.00) & f(0.01) & \ldots & f(0.09) \\ f(0.10) & f(0.11) & \ldots & f(0.19) \\ \ldots & \ldots & \ldots & \ldots \\ f(1.90) & f(1.91) & \ldots & f(1.99) \end{array}$$

Escriba una función que extraiga de la tabla `NORMAL.TXT` creada el valor correspondiente en vez de calcularlo.

4. Se ha efectuado un examen de tipo test, con 20 preguntas, a los alumnos de un grupo. En el archivo `EXAMENES.TXT` se hallan las respuestas, con arreglo al siguiente formato: en los primeros 25 caracteres se consigna el nombre, a continuación sigue un espacio en blanco y, en los 20 siguientes, letras de la 'A' a la 'E', correspondientes a las respuestas. En la primera línea del archivo `SOLUCION.TXT` se hallan las soluciones correctas consignadas en las 20 primeras casillas.

   Se considera que una respuesta válida suma cuatro puntos, una incorrecta resta un punto y un carácter distinto a los posibles anula esa pregunta, no puntuando positiva ni negativamente. La nota se halla usando la expresión `Round(puntuación/8)`.

   Escriba un programa que confeccione otro archivo `RESULT.TXT` con la lista de aprobados junto con la puntuación obtenida.

5. (a) Partiendo del conjunto de los primos entre 2 y 256 creado mediante el algoritmo de la criba de Eratóstenes (véase el apartado 11.3.3), escriba un programa que los guarde en un archivo de disco, `PRIMOS.TXT`.

   (b) Defina una función que compruebe si un número menor que 256 es primo, simplemente consultando la tabla `PRIMOS.TXT`. (Al estar ordenada ascendentemente, con frecuencia será innecesario llegar al final de la misma.)

   (c) Defina una función que compruebe si un número n entre cien y 256^2 es primo, tanteando como posibles divisores los números de la tabla `PRIMOS.TXT` entre 2 y $\lfloor\sqrt{n}\rfloor$ que sea necesario.

   (d) Integre los apartados anteriores que convengan en un programa que genere los primos menores que 256^2.

6. Escriba un programa que convierta en archivo de texto, de nombre dado, las líneas introducidas desde el `input`, de forma similar al funcionamiento de la orden `copy ... con:` del DOS. Escriba igualmente un programa que muestre un archivo de texto, de nombre dado, por pantalla, tal como hace la orden `type` del DOS.

7. Escriba un subprograma que reciba dos archivos de texto y los mezcle, carácter a carácter, en un tercer archivo de texto. Si alguno de los archivos origen terminara antes que el otro, el subprograma añadirá al archivo destino lo que quede del otro archivo origen.

8. Añada al programa del ejercicio 6 del capítulo anterior una opción que permita almacenar y recuperar los datos de los productos en un archivo de texto o con tipo.

Capítulo 15

Algoritmos de búsqueda y ordenación

Una vez vistos los distintos tipos de datos que el programador puede definir, se presentan en este capítulo dos de las aplicaciones más frecuentes y útiles de los tipos de datos definidos por el programador: la búsqueda y la ordenación. En particular, estas aplicaciones afectan directamente a los dos tipos de datos estudiados hasta ahora que permiten el almacenamiento de datos: los arrays, para datos no persistentes en el tiempo, y los archivos, para datos que deben ser recordados de una ejecución a otra de un determinado programa.

15.1 Algoritmos de búsqueda en arrays

Es evidente que, si tenemos datos almacenados, es interesante disponer de algún mecanismo que permita saber si un cierto dato está entre ellos, y, en caso afirmativo, localizar la posición en que se encuentra para poder trabajar con

él. Los mecanismos que realizan esta función son conocidos como algoritmos de búsqueda.

El problema que se plantea a la hora de realizar una búsqueda (concretamente en un array) puede ser enunciado de la siguiente forma:

Supongamos que tenemos un vector `v` con `n` elementos (los índices son los `1...n`) y pretendemos construir una función `Busqueda` que encuentre un índice `i` de tal forma que `v[i] = elem`, siendo `elem` el elemento que se busca. Si no existe tal índice, la función debe devolver un cero, indicando así que el elemento `elem` buscado no está en el vector `v`. En resumen,

$$\texttt{Busqueda} : \mathcal{V}_n(\texttt{tElem}) \times \texttt{tElem} \longrightarrow \{0, 1, \ldots, n\}$$

de forma que

$$\texttt{Busqueda}(\texttt{v}, \texttt{elem}) = \begin{cases} \texttt{i} \in \{1, \ldots, n\} & \text{si existe } \texttt{i} \text{ tal que } \texttt{elem} = \texttt{v}_\texttt{i} \\ 0 & \text{en otro caso} \end{cases}$$

En todo el capítulo se definen los elementos del vector con el tipo `tElem`:

```
const
  N = 100; {tamaño del vector}
type
  tIntervalo = 0..N;
  tVector = array [1..N] of tElem;
```

Para simplificar, digamos por ahora que `tElem` es un tipo ordinal, siendo por tanto comparables sus valores.

Los algoritmos más usuales que se pueden desarrollar para tal fin son los algoritmos de búsqueda secuencial y búsqueda binaria.[1]

15.1.1 Búsqueda secuencial

La búsqueda secuencial consiste en comparar secuencialmente el elemento deseado con los valores contenidos en las posiciones 1, ..., n hasta que, o bien encontremos el índice `i` buscado, o lleguemos al final del vector sin encontrarlo, concluyendo que el elemento buscado no está en el vector.

La búsqueda secuencial es un algoritmo válido para un vector cualquiera sin necesidad de que esté ordenado. También se puede aplicar con muy pocas

[1] Conocida también con los nombres de búsqueda dicotómica o por bipartición.

variaciones a otras estructuras secuenciales, como, por ejemplo, a los archivos (véase el apartado 15.3).

El primer nivel en el diseño de la función `BusquedaSec` puede ser:

```
ind:= 0;
Buscar elem en v;
Devolver el resultado de la función
```

Refinando *Buscar* `elem` *en* `v`, se tiene:

```
repetir
  ind:= ind + 1
hasta que v[ind] = elem o ind = n
```

Por último, refinando *Devolver el resultado de la función* se obtiene:

```
si v[ind] = elem entonces
  BusquedaSec:= ind
si no
  BusquedaSec:= 0
```

Una posible implementación de la función `BusquedaSec` siguiendo el esquema de este algoritmo secuencial es:

```
function BusquedaSec(v: tVector; elem: tElem): tIntervalo;
  {Dev. 0 (si elem no está en v) ó i (si v[i] = elem)}
  var
    i: tIntervalo;
begin
  i:= 0; {se inicia el contador}
  repeat
      {Inv.: ∀ j, 0 ≤ j ≤ i ⇒ v[j] ≠ elem}
    i:= i + 1
  until (v[i] = elem) or (i = N);
    {v[i] = elem
  if v[i] = elem then {se ha encontrado el elemento elem}
    BusquedaSec:= i
  else
    BusquedaSec:= 0
end; {BusquedaSec}
```

15.1.2 Búsqueda secuencial ordenada

El algoritmo de búsqueda secuencial puede ser optimizado si el vector `v` está ordenado (supongamos que de forma creciente). En este caso, la búsqueda secuencial desarrollada anteriormente es ineficiente, ya que, si el elemento buscado `elem` no se encuentra en el vector, se tendrá que recorrer todo el vector, cuando se sabe que si se llega a una componente con valor mayor que `elem`, ya no se encontrará el valor buscado.

Una primera solución a este nuevo problema sería modificar la condición de salida del bucle **repeat** cambiando `v[i]=elem` por `v[i]>=elem`, debido a que el vector se encuentra ordenado de forma creciente:

```
function BusquedaSecOrd(v: tVector; elem: tElem): tIntervalo;
  {PreC.: v está ordenado crecientemente}
  {Dev. 0 (si elem no está en v) ó i (si v[i] = elem)}
  var
    i: tIntervalo;
begin
  i:= 0;
  repeat
      {Inv.: ∀ j, 0 ≤ j ≤ i, ⇒ v[j] ≠ elem}
    i:= i + 1
  until (v[i]>=elem) or (i=N);
    {v[i]=elem
  if v[i] = elem then {se ha encontrado el elemento elem}
    BusquedaSecOrd:= i
  else
    BusquedaSecOrd:= 0
end; {BusquedaSecOrd}
```

Esta solución también se puede aplicar a archivos secuenciales ordenados (véase el apartado 15.3).

15.1.3 Búsqueda binaria

El hecho de que el vector esté ordenado se puede aprovechar para conseguir una mayor eficiencia en la búsqueda planteando el siguiente algoritmo: comparar `elem` con el elemento central; si `elem` es ese elemento ya hemos terminado, en otro caso buscamos en la mitad del vector que nos interese (según sea `elem` menor o mayor que el elemento mitad, buscaremos en la primera o segunda mitad del vector, respectivamente). Posteriormente, si no se ha encontrado el elemento repetiremos este proceso comparando `elem` con el elemento central del subvector seleccionado, y así sucesivamente hasta que o bien encontremos el valor `elem` o bien podamos concluir que `elem` no está (porque el subvector de búsqueda está

vacío). Este algoritmo de búsqueda recibe el nombre de búsqueda binaria, ya que va dividiendo el vector en dos subvectores de igual tamaño.

Vamos ahora a realizar una implementación de una función siguiendo el algoritmo de búsqueda binaria realizando un diseño descendente del problema, del cual el primer refinamiento puede ser:

Asignar valores iniciales `extInf, extSup, encontrado;`
Buscar `elem` *en* `v[extInf..extSup];`
Devolver el resultado de la función

Refinando *Asignar valores iniciales* se tiene:

```
extInf:= 1;
extSup:= N;
  {se supone que N es el tamaño del array inicial}
encontrado:= False;
```

En un nivel más refinado de *Buscar* `elem` *en* `v[extInf..extSup]` se tiene:

mientras el vector no sea vacío y
no se ha encontrado el valor `c` *hacer*
  calcular el valor de `posMed`*;*
  si `v[posMed] = elem` *entonces*
    actualizar el valor de `encontrado`
  si no
    actualizar los valores `extInf o extSup` *según donde esté* `elem`

Refinando *devolver el resultado de la función* obtenemos:

si se ha encontrado el valor entonces
  `BusquedaBinaria:= posMed;`
si no
  `BusquedaBinaria:= 0;`

Con todo esto, una posible implementación sería:

```
function BusquedaBinaria(v: tVector; elem: tElem): tIntervalo;
  {PreC.: v está ordenado crecientemente}
  {Dev. 0 (si elem no está en v) ó i (si v[i] = elem)}
  var
    extInf, extSup, {extremos del intervalo}
    posMed: tIntervalo; {posición central del intervalo}
    encontrado: boolean;
```

```
begin
  extInf:= 1;
  extSup:= N;
  encontrado:= False;
  while (not encontrado) and (extSup >= extInf) do begin
      {Inv.:  si elem está en v, ⇒ v[extInf] ≤ elem ≤ v[extSup]}
    posMed:= (extSup + extInf) div 2;
    if elem = v[posMed] then
      encontrado:= True
    else if elem > v[posMed] then
        {se actualizan los extremos del intervalo}
      extInf:= posMed + 1
    else
      extSup:= posMed - 1
  end; {while}
  if encontrado then
    BusquedaBinaria:= palMed
  else
    BusquedaBinaria:= 0
end; {BusquedaBinaria}
```

La necesidad de acceder de forma directa a las componentes intermedias no permite la aplicación de este tipo de soluciones a archivos secuenciales.

15.2 Ordenación de arrays

En muchas situaciones se necesita tener ordenados, según algún criterio, los datos con los que se trabaja para facilitar su tratamiento. Así, por ejemplo, en un vector donde se tengan almacenados los alumnos de cierta asignatura junto con la calificación obtenida, sería interesante ordenar los alumnos por orden alfabético, o bien, ordenar el vector según la calificación obtenida para poder sacar una lista de "Aprobados" y "Suspensos".

En este apartado se presentan los algoritmos de ordenación más usuales, de los muchos existentes. Su objetivo común es resolver el problema de ordenación que se enuncia a continuación:

Sea `v` un vector con `n` componentes de un mismo tipo, `tElem`. Definimos la función ordenación como:

$$\texttt{ordenación} : \mathcal{V}_n(\texttt{tElem}) \longrightarrow \mathcal{V}_n(\texttt{tElem})$$

de tal forma que `ordenación(v) = v'` donde $\texttt{v'=(v'}_1\texttt{,..., v'}_n\texttt{)}$ es una permutación de $\texttt{v=(v}_1\texttt{,..., v}_n\texttt{)}$ tal que $\texttt{v'}_1 \leq \texttt{v'}_2 \leq \ldots \leq \texttt{v'}_n$ donde $\leq$ es la relación de orden elegida para clasificar los elementos de `v`.

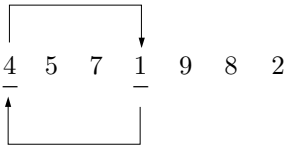
En todos los algoritmos que se van a estudiar a continuación, se supondrá que la ordenación se llevará a cabo en forma creciente. Es obvio que las modificaciones que se tendrían que realizar para ordenar un vector con otra relación de orden son inmediatas y se dejan para el lector.

15.2.1 Selección directa

Este algoritmo resuelve el problema de la ordenación recorriendo el vector y seleccionando en cada recorrido el menor elemento para situarlo en su lugar correspondiente. El esquema básico del algoritmo de selección directa es:

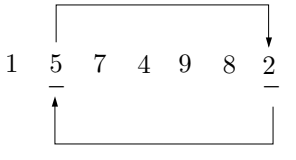
1. Se sitúa en $\mathtt{v}_1$ el menor valor entre $\mathtt{v}_1, \ldots, \mathtt{v}_n$. Para ello se intercambian los valores de $\mathtt{v}_1$ y $\mathtt{v}_\mathtt{m}$ siendo $\mathtt{v}_\mathtt{m} = \min\limits_{k=1,\ldots,n} \{v_k\}$.

   Por ejemplo, para el vector (4, 5, 7, 1, 9, 8, 2), este primer paso produce el intercambio representado en la siguiente figura:



2. Se sitúa en $\mathtt{v}_2$ el menor valor entre $\mathtt{v}_2, \ldots, \mathtt{v}_n$. Para ello se intercambian los valores de $\mathtt{v}_2$ y $\mathtt{v}_\mathtt{m}$ siendo $\mathtt{v}_\mathtt{m} = \min\limits_{k=2,\ldots,n} \{v_k\}$.

   En el ejemplo, se produce el intercambio:



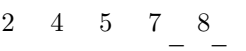
...

(j-1). Se sitúa en $\mathtt{v}_{\mathtt{j}-1}$ el menor valor entre $\mathtt{v}_{\mathtt{j}-1}, \ldots, \mathtt{v}_n$. Para ello se intercambian los valores de $\mathtt{v}_{\mathtt{j}-1}$ y $\mathtt{v}_\mathtt{m}$ siendo $\mathtt{v}_\mathtt{m} = \min\limits_{k=j-1,\ldots,n} \{v_k\}$.

...

(n-1). Se sitúa en $\mathtt{v}_{\mathtt{n}-1}$ el menor valor entre $\mathtt{v}_{\mathtt{n}-1}$ y $\mathtt{v}_\mathtt{n}$. Para ello se intercambian los valores de $\mathtt{v}_{\mathtt{n}-1}$ y $\mathtt{v}_\mathtt{n}$ si es necesario.

La situación final en el ejemplo es representada en la siguiente figura:



El primer nivel de diseño del algoritmo es:

para cada `i` *entre 1 y* `n - 1` *hacer*
  Colocar en $\texttt{v}_\texttt{i}$ *el menor entre* $\texttt{v}_\texttt{i}, \ldots, \texttt{v}_n$*;*
Devolver `v`*, ya ordenado*

donde *Colocar en* $\texttt{v}_\texttt{i}$ *el menor entre* $\texttt{v}_\texttt{i}, \ldots, \texttt{v}_n$ es:

```
valMenor:= v[i];
posMenor:= i;
para cada j entre i + 1 y n hacer
  si v[j] < valMenor entonces
    valMenor:= v[j];
    posMenor:= j
  fin {si}
fin {para}
Intercambiar v[i] con v[PosMenor];
```

Por lo tanto, una posible implementación de ordenación de un vector, siguiendo el algoritmo de selección directa, podría ser la siguiente:

```
procedure SeleccionDirecta(var v: tVector);
  {Efecto: se ordena v ascendentemente}
  var
    i, j, posMenor: tIntervalo;
    valMenor, aux: integer;
begin
  for i:= 1 to N-1 do begin
      {Inv.: ∀ j, 1 ≤ j ≤ i − 1, ⇒ v[j] ≤ v[j+1]
       y además todos los v[i]...v[N] son mayores que v[i-1]}
    valMenor:= v[i];
      {se dan valores iniciales}
    posMenor:= i;
    for j:= i + 1 to n do
      if v[j] < valMenor then begin
          {se actualiza el nuevo valor menor y la
          posición donde se encuentra}
        valMenor:= v[j];
        posMenor:= j
      end {if}
```

```
      if posMenor <> i then begin
          {Si el menor no es v[i], se intercambian los valores}
        aux:= v[i];
        v[i]:= v[posMenor];
        v[posMenor]:= aux
      end {if}
    end; {for i}
  end; {SeleccionDirecta}
```

15.2.2 Inserción directa

Este algoritmo recorre el vector v insertando el elemento v_i en su lugar correcto entre los ya ordenados $v_1, \ldots, v_{i-1}$. El esquema general de este algoritmo es:

1. Se considera v_1 como primer elemento.
2. Se inserta v_2 en su posición correspondiente en relación a v_1 y v_2.
3. Se inserta v_3 en su posición correspondiente en relación a $v_1, \ldots, v_3$.

...

i. Se inserta v_i en su posición correspondiente en relación a $v_1, \ldots, v_i$.

...

n. Se inserta v_n en su posición correspondiente en relación a $v_1, \ldots, v_n$.

En el diagrama de la figura 15.1 se muestra un ejemplo de aplicación de este algoritmo.

Atendiendo a esta descripción, el diseño descendente de este algoritmo tiene como primer nivel el siguiente:

para cada `i` *entre 2 y* `N` *hacer*
  Situar `v[i]` *en su posición ordenada respecto a* `v[1]`,..., `v[i-1]`

Donde *Situar* `v[i]` *en su posición ordenada respecto a* `v[1]`,..., `v[i-1]` puede refinarse de la siguiente forma:

Localizar la posición `j` *entre* `1` *e* `i-1` *correspondiente a* `v[i]`;
Desplazar una posición las componentes `v[j+1]`,..., `v[i-1]`;
`v[j]:= v[i]`

Por lo tanto, una implementación de este algoritmo será:

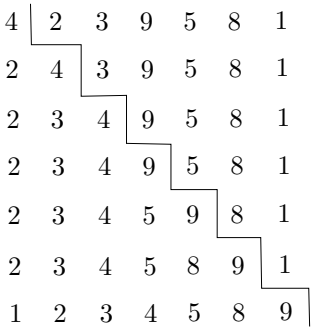


Figura 15.1.

```
procedure InsercionDirecta(var v:  tVector);
  {Efecto:  se ordena v ascendentemente}
  var
    i, j:  tIntervalo;
    aux:  tElem;
begin
  for i:= 2 to N do begin
      {Inv.:  ∀ j, 1 ≤ j < i, ⇒ v[j] ≤ v[j+1]}
    aux:= v[i];
      {se dan los valores iniciales}
    j:= i - 1;
    while (j >= 1) and (v[j] > aux) do begin
      v[j+1]:= v[j];
        {Desplazamiento de los valores mayores que v[i]}
      j:= j-1
    end; {while}
    v[j+1]:= aux
  end {for}
end; {InsercionDirecta}
```

15.2.3 Intercambio directo

El algoritmo por intercambio directo recorre el vector buscando el menor elemento desde la última posición hasta la actual y lo sitúa en dicha posición. Para ello, se intercambian valores vecinos siempre que estén en orden decreciente. Así se baja, mediante sucesivos intercambios, el valor menor hasta la posición deseada. Más concretamente, el algoritmo consiste en:

1. Situar el elemento menor en la primera posición. Para ello se compara el último elemento con el penúltimo, intercambiando sus valores si están

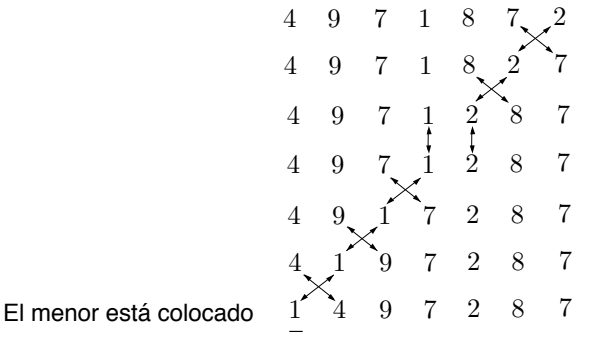


Figura 15.2.

en orden decreciente. A continuación se comparan el penúltimo elemento con el anterior, intercambiándose si es necesario, y así sucesivamente hasta llegar a la primera posición. En este momento se puede asegurar que el elemento menor se encuentra en la primera posición.

Así, para el vector (4, 9, 7, 1, 8, 7, 2), se realiza el proceso representado en la figura 15.2.

2. Situar el segundo menor elemento en la segunda posición. Para ello se procede como antes finalizando al llegar a la segunda posición, con lo que se sitúa el elemento buscado en dicha posición.

   En el ejemplo, se obtiene en este paso el vector (1, 2, 4, 9, 7, 7, 8).

... Se repite el proceso para las posiciones intermedias.

(n-1). Se comparan los dos últimos valores, intercambiándose si están en orden decreciente, obteniendose así el vector ordenado. En el caso del ejemplo se obtiene el vector (1, 2, 4, 7, 7, 8, 9).

El seudocódigo correspondiente al primer nivel de diseño de este algoritmo es:

para `i` *entre* `1` *y* `n-1` *hacer*
  Desplazar el menor valor desde $\texttt{v}_{\texttt{n}}$ *hasta* $\texttt{v}_{\texttt{i}}$*, intercambiando pares vecinos, si es necesario*
Devolver `v`*, ya ordenado*

Por lo tanto, una implementación del algoritmo puede ser:

```
procedure OrdenacionPorIntercambio(var v:  tVector);
  {Efecto:  se ordena v ascendentemente}
  var
    i, j:  tIntervalo;
    aux:  tElem;
begin
  for i:= 1 to N-1 do
      {Inv.:  ∀ j, 1 ≤ j < i, ⇒ v[j] ≤ v[k], ∀ k tal que j ≤ k < N}
    for j:= N downto i + 1 do
        {Se busca el menor desde atrás y se sitúa en v_i}
      if v[j-1] > v[j] then begin {intercambio}
        aux:= v[j];
        v[j]:= v[j-1];
        v[j-1]:= aux
      end {if}
end; {OrdenacionPorIntercambio}
```

15.2.4 Ordenación rápida (*Quick Sort*)

El algoritmo de *ordenación rápida* [2] debido a Hoare, consiste en dividir el vector que se desea ordenar en dos bloques. En el primer bloque se sitúan todos los elementos del vector que son menores que un cierto valor de `v` que se toma como referencia (valor pivote), mientras que en el segundo bloque se colocan el resto de los elementos, es decir, los que son mayores que el valor pivote. Posteriormente se ordenarán (siguiendo el mismo proceso) cada uno de los bloques, uniéndolos una vez ordenados, para formar la solución. En la figura 15.3 se muestran gráficamente las dos fases de ordenación.

Evidentemente, la condición de parada del algoritmo se da cuando el bloque que se desea ordenar esté formado por un único elemento, en cuyo caso, obviamente, el bloque ya se encuentra ordenado.

También se puede optar por detener el algoritmo cuando el número de elementos del bloque sea suficientemente pequeño (generalmente con un número aproximado de 15 elementos), y ordenar éste siguiendo alguno de los algoritmos vistos anteriormente (el de inserción directa, por ejemplo).

El número elegido de 15 elementos es orientativo. Se debería elegir dicha cantidad mediante pruebas de ensayo para localizar el valor óptimo.

Aunque cualquier algoritmo de ordenación visto anteriormente sea "más lento" (como veremos en el capítulo de complejidad algorítmica) que el *Quick Sort*, este último pierde gran cantidad de tiempo en clasificar los elementos en

[2] *Quick Sort* en inglés.

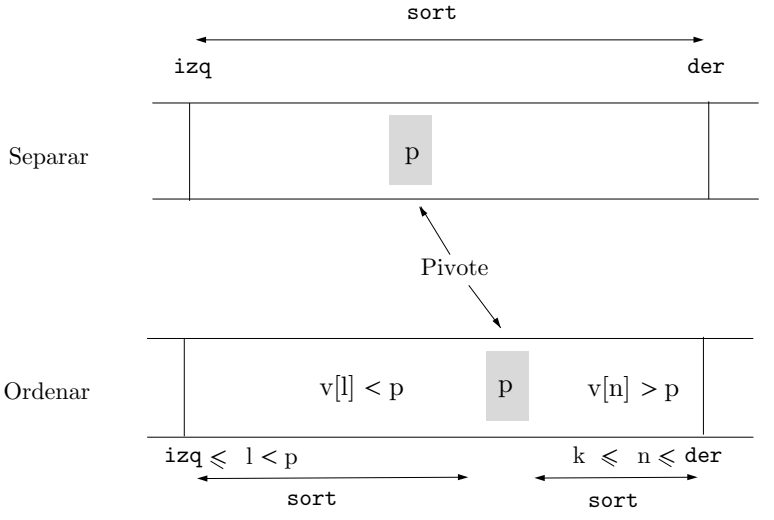


Figura 15.3.

los dos bloques, por lo que, cuando el número de elementos es pequeño, no es "rentable" utilizar *Quick Sort*.

En este apartado vamos a desarrollar el algoritmo *Quick Sort* con la primera condición de parada, dejando al lector el desarrollo de la segunda versión.

Este algoritmo sigue el esquema conocido con el nombre de *divide y vencerás* (véase el apartado 20.2) que, básicamente, consiste en subdividir el problema en dos iguales pero de menor tamaño para posteriormente combinar las dos soluciones parciales obtenidas para producir la solución global.

El seudocódigo correspondiente al primer nivel en el diseño descendente del algoritmo *Quick Sort* es:

si `v` *es de tamaño 1 entonces*
  `v` *ya está ordenado*
si no
  Dividir `v` *en dos bloques* `A` *y* `B`
  con todos los elementos de `A` *menores que los de* `B`
fin {`si`}
Ordenar `A` *y* `B` *usando Quick Sort*
Devolver `v` *ya ordenado como concatenación*
de las ordenaciones de `A` *y de* `B`

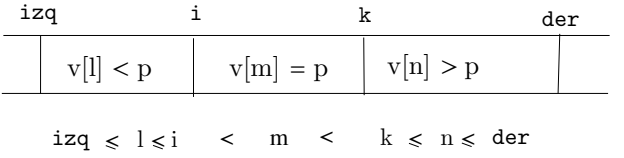


Figura 15.4.

Donde *Dividir* `v` *en dos bloques* `A` *y* `B` se puede refinar en:

Elegir un elemento `p` *(pivote) de* `v`
para cada elemento del vector hacer
  si elemento `<` `p` *entonces*
    Colocar elemento en `A`*, el subvector con los elementos de* `v`
    menores que `p`
  en otro caso
    Colocar elemento en `B`*, el subvector con los elementos de* `v`
    mayores que `p`

De cara a la implementación, y por razones de simplicidad y ahorro de memoria, es preferible situar los subvectores sobre el propio vector original `v` en lugar de generar dos nuevos arrays.

Con todo lo anterior, una implementación de este algoritmo podría ser:

```
procedure QuickSort(var v:  tVector);
  {Efecto:  se ordena v ascendentemente}
  procedure SortDesdeHasta(var v: tVector; izq,der: tIntervalo);
    {Efecto:  v[izq..der] está ordenado ascendentemente}
    var
      i,j:  tIntervalo;
      p,aux:  tElem;
  begin
    i:= izq;
    j:= der;
      {se divide el vector v[izq..der] en dos trozos eligiendo como
       pivote p el elemento medio del array}
    p:= v[(izq + der) div 2];
      {si i >= d el subvector ya está ordenado}
      {Inv.:  ∀ s, izq ≤ s < i, ⇒ v[s] < p
       y ∀ t tal que j < t ≤ der ⇒ v[s] > p}
```

```
    while i < j do begin
        {se reorganizan los dos subvectores}
      while v[i] < p do
        i:= i + 1;
      while p < v[j] do
        j:= j - 1;
      if i <= j then begin
          {intercambio de elementos}
        aux:= v[i];
        v[i]:= v[j];
        v[j]:= aux;
          {ajuste de posiciones}
        i:= i + 1;
        j:= j - 1
      end {if}
    end;  {while}
    if izq < j then
      SortDesdeHasta(v,izq,j);
    if i < der then
      SortDesdeHasta(v,i,der)
  end; {SortDesdeHasta}

begin {QuickSort}
  SortDesdeHasta(v,1, n)
end; {QuickSort}
```

Para facilitar la comprensión del método, se ha realizado un ejemplo de su funcionamiento provocando algunas llamadas sobre el vector inicial `v= [0, 3, 86, 20, 27]` y mostrando el vector tras cada llamada:

```
                              v = [0, 3, 86, 20, 27]
        SortDesdeHasta(v,1,5)  v = [0, 3, 27, 20, 86]
        SortDesdeHasta(v,1,4)  v = [0, 3, 27, 20, 86]
        SortDesdeHasta(v,1,4)  v = [0, 3, 27, 20, 86]
```

A propósito de este algoritmo, es necesaria una observación sobre su eficiencia. El mejor rendimiento se obtiene cuando el pivote elegido da lugar a dos subvectores de igual tamaño, dividiéndose así el vector en dos mitades. Este algoritmo también proporciona un coste mínimo cuando los elementos del vector se distribuyen aleatoriamente (equiprobablemente).

Sin embargo, puede ocurrir que los elementos estén dispuestos de forma que, cada vez que se elige el pivote, todos los elementos queden a su izquierda o todos a su derecha. Entonces, uno de los dos subvectores es vacío y el otro carga con todo el trabajo. En este caso, el algoritmo da un pésimo rendimiento, comparable a

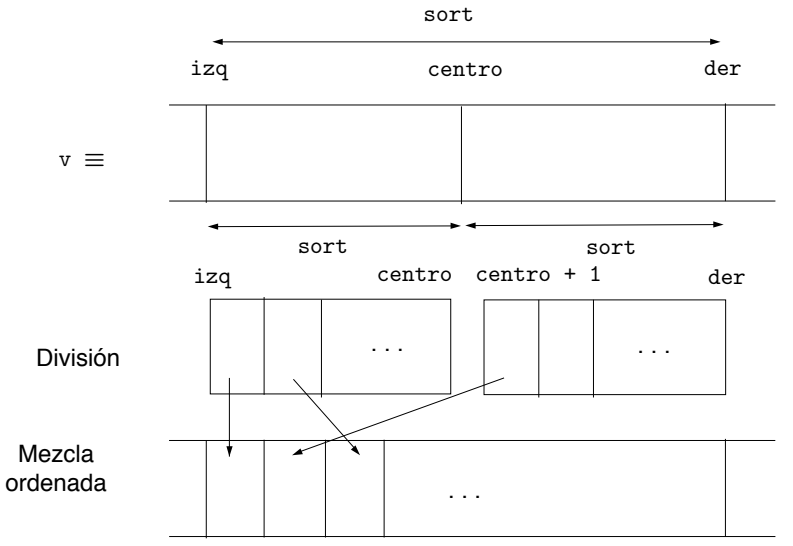


Figura 15.5.

los algoritmos de selección, inserción e intercambio. Un ejemplo de esta situación se tiene cuando el vector por ordenar tiene los elementos inicialmente dispuestos en orden descendente y el pivote elegido es, precisamente, el primer (o el último) elemento.

15.2.5 Ordenación por mezcla (*Merge Sort*)

Al igual que *Quick Sort*, el algoritmo de ordenación *Merge Sort* va a utilizar la técnica divide y vencerás. En este caso la idea clave del algoritmo consiste en dividir el vector `v` en dos subvectores `A` y `B` (de igual tamaño, si es posible), sin tener en cuenta ningún otro criterio de división.

Posteriormente se mezclarán ordenadamente las soluciones obtenidas al ordenar `A` y `B` (aplicando nuevamente, a cada uno de los subvectores, el algoritmo *Merge Sort*). En la figura 15.5 se muestra un esquema de este algoritmo.

Para *Merge Sort* también se pueden considerar las condiciones de parada descritas en el apartado anterior para el algoritmo *Quick Sort*, desarrollándose a continuación el diseño descendente y la implementación para el primer caso de parada (cuando se llega a subvectores de longitud 1). Se deja como ejercicio para el lector el desarrollo de la implementación de *Merge Sort* con la segunda condición de parada, es decir, la combinación de *Merge Sort* con otro algoritmo de ordenación cuando se llega a subvectores de una longitud dada.

El algoritmo *Merge Sort* puede esbozarse como sigue:

si `v` *es de tamaño 1 entonces*
  `v` *ya está ordenado*
si no
  Dividir `v` *en dos subvectores* `A` *y* `B`
fin {`si`}
Ordenar `A` *y* `B` *usando Merge Sort*
Mezclar las ordenaciones de `A` *y* `B` *para generar el vector ordenado.*

En este caso, el paso *Dividir* `v` *en dos subvectores* `A` *y* `B` consistirá en:

Asignar a `A` *el subvector* $[v_1, \ldots, v_{n \operatorname{div} 2}]$
Asignar a `B` *el subvector* $[v_{n \operatorname{div} 2+1}, \ldots, v_n]$

mientras que *Mezclar las ordenaciones de* `A` *y* `B` consistirá en desarrollar un procedimiento (que llamaremos `Merge`) encargado de ir entremezclando adecuadamente las componentes ya ordenadas de `A` y de `B` para obtener el resultado buscado.

De acuerdo con este diseño se llega a la implementación que se muestra a continuación. Al igual que en la implementación de *Quick Sort*, se sitúan los subvectores sobre el propio vector original `v` en lugar de generar dos nuevos arrays.

```
procedure MergeSort(var vector:  tVector);
  {Efecto:  se ordena vector ascendentemente}
  procedure MergeSortDesdeHasta(var v:  vector; izq, der:  integer);
    {Efecto:  se ordena v[izq..der] ascendentemente}
    var
      centro :  tIntervalo;

    procedure Merge(vec:  tVector; iz, ce, de:  tIntervalo;
                    var w:  tVector);
      {Efecto:  w := mezcla ordenada de los subvectores v[iz..ce] y
       v[ce+1..de]}
      var
        i,j,k:  1..N;
    begin {Merge}
      i:= iz;
      j:= ce + 1;
      k:= iz;
        {k recorre w, vector que almacena la ordenación total}
      while (i <= ce) and (j <= de) do begin
          {Inv.:  ∀ m, iz ≤ m < k, ⇒ w[m] ≤ w[m+1]}
        if vec[i] < vec[j] then begin
```

```
            w[k]:= vec[i];
            i:= i + 1
          end {Then}
          else begin
            w[k]:= vec[j];
            j:= j + 1
          end; {Else}
          k:= k + 1;
        end; {While}
        for k:= j to de do
          w[k]:= vec[k]
        for k:= i to ce do
          w[k+de-ce]:= vec[k]
      end; {Merge}

    begin {MergeSortDesdeHasta}
      centro:= (izq + der) div 2;
      if izq < centro then
        MergeSortDesdeHasta(v, izq, centro);
      if centro < der then
        MergeSortDesdeHasta(v, centro+1, der);
      Merge(v, izq, centro, der, v)
    end; {MergeSortDesdeHasta}

  begin {MergeSort}
    MergeSortDesdeHasta(vector, 1, N)
  end; {MergeSort}
```

Para facilitar la comprensión del método se ha realizado un ejemplo, provocando algunas llamadas sobre el vector inicial `v= [8, 5, 7, 3]` y mostrando el vector tras cada llamada:

```
                                    v = [8, 5, 7, 3]
MergeSortDesdeHasta(v,1,2)          v = [5, 8, 7, 3]
MergeSortDesdeHasta(v,3,4)          v = [5, 8, 3, 7]
Merge(v,1,2,4,v)                    v = [3, 5, 7, 8]
```

15.2.6 Vectores paralelos

Supóngase que se desea ordenar un vector `vectFich` de la forma

array`[1..N]` **of** `tFicha`

siendo el tipo `tFicha` de un gran tamaño.

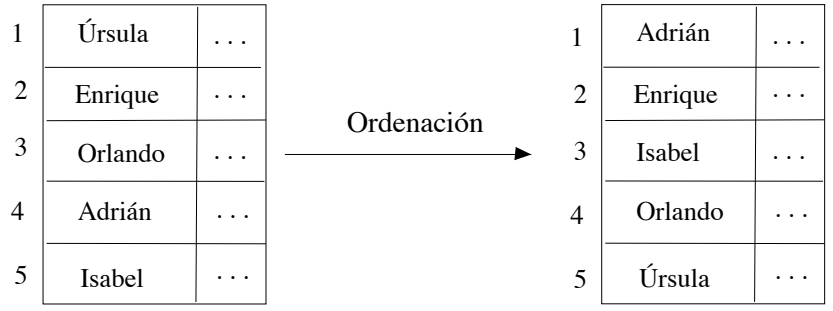


Figura 15.6.

Una posibilidad interesante consiste en crear otro vector `vectPosic` de la forma

array`[1..N]` **of** `[1..N]`

cuyas componentes refieren una ficha completa, mediante su posición, siendo inicialmente

`vectPosic[i] = i` $\quad \forall i \in \{1, \ldots N\}$

El interés de esta representación indirecta reside en aplicar este método a cualquiera de los algoritmos de ordenación estudiados, con la ventaja de que la ordenación se puede efectuar intercambiando las posiciones en vez de los registros (extensos) completos.

Por ejemplo, supongamos que el tipo de datos `tFicha` es un registro con datos personales, entre los que se incluye el nombre. La figura 15.6 muestra el vector `vectFich` antes y después de la ordenación.

Si no se emplea el método de los vectores paralelos, el algoritmo de ordenación empleado debería incluir instrucciones como las siguientes:

```
if vectFich[i].nombre > vectFich[j].nombre then begin
    {Intercambiar vectFich[i], vectFich[j]}
  elemAux:= vectFich[i];
  vectFich[i]:= vectFich[j];
  vectFich[j]:= elemAux
end
```

Como se puede comprobar, en las instrucciones de intercambio es necesario mantener tres copias de elementos del tipo `tElemento`. Sin embargo, si se utiliza un vector paralelo (como se muestra en la figura 15.7), sólo se producen cambios en el vector de posiciones. El fragmento de código correspondiente a los intercambios quedaría como sigue:

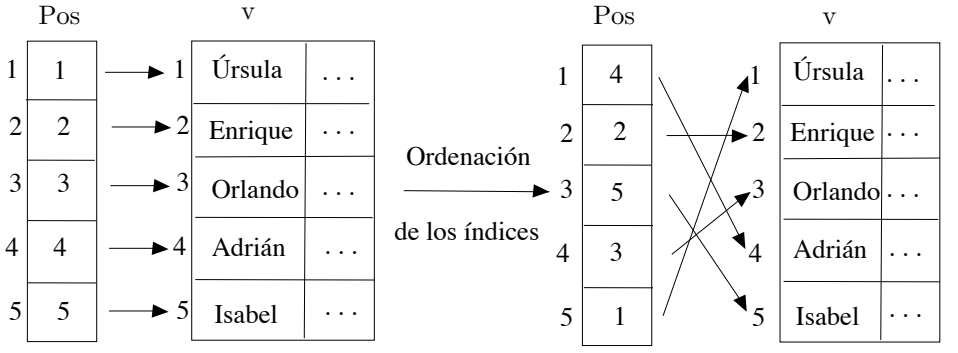


Figura 15.7.

```
if vectFich[vectPosic[i]].nombre > vectFich[vectPosic[j]].nombre
then begin
    {Intercambiar vectPosic[i], vectPosic[j] ∈ [1..N]}
  posAux:= vectPosic[i];
  vectPosic[i]:= vectPosic[j];
  vectPosic[j]:= posAux
end
```

Así se manipulan únicamente tres copias de posiciones, que son de tipo integer, con el consiguiente ahorro de tiempo y espacio, ya que es más eficiente intercambiar dos índices que dos elementos (siempre y cuando éstos sean grandes).

Además, el método de los vectores paralelos tiene la ventaja añadida de que permite mantener varios índices, que en el ejemplo permitirían realizar ordenaciones por nombre, DNI, etc.

Finalmente, también conviene indicar que la idea de manejar las posiciones en lugar de los elementos mismos se explota ampliamente en programación dinámica (véanse los capítulos 16 y 17).

15.3 Algoritmos de búsqueda en archivos secuenciales

La variedad de algoritmos de búsqueda en archivos se ve muy restringida en Pascal por la limitación a archivos de acceso secuencial. Este hecho obliga a que la localización del elemento buscado se haga examinando las sucesivas

componentes del archivo. Por tanto, la estrategia de búsqueda será la misma que se emplea en los algoritmos de búsqueda secuencial en arrays (véanse los apartados 15.1.1 y 15.1.2).

En este breve apartado nos limitaremos a mostrar la adaptación directa de estos algoritmos de búsqueda para archivos arbitrarios y para archivos ordenados.

15.3.1 Búsqueda en archivos arbitrarios

Supongamos que se dispone de un archivo definido de la siguiente forma:

```
type
  tElem = record
    clave: tClave;
    resto de la información
  end;
  tArchivoElems = file of tElem;
```

Con esto, el procedimiento de búsqueda queda como sigue (por la simplicidad de la implementación se han obviado los pasos previos de diseño y algunos aspectos de la corrección):

```
procedure Buscar(var f: tArchivoElems; c: tClave; var elem: tElem;
                 var encontrado: boolean);
  {PostC: si c está en f entonces encontrado = True y elem es
   el elemento buscado; en otro caso encontrado = False}
begin
  encontrado:= False;
  while not EoF(f) and not encontrado do begin
    Read(f, elem);
    if c = elem.clave then
      encontrado:= True
  end {while}
end; {Buscar}
```

15.3.2 Búsqueda en archivos ordenados

Al igual que ocurre con la búsqueda secuencial en arrays, el algoritmo puede mejorarse si las componentes del archivo están ordenadas por sus claves. En este caso, se puede evitar recorrer inútilmente la parte final del archivo si hemos detectado una clave mayor que la buscada, ya que entonces tendremos la certeza de que el elemento no está en el archivo. La modificación en el código es muy sencilla, puesto que basta con variar la condición de salida del bucle while para que contemple la ordenación del archivo. El nuevo procedimiento es el siguiente:

```
procedure BuscarOrd(var f: tArchivoElems; c: clave;
                    var elem: tElem; var encontrado: boolean);
  {PostC.: si c está en f entonces encontrado = True y elem es
  el elemento buscado; en otro caso encontrado = False}
  var
    ultimaClave: clave;

begin
  encontrado:= False;
  ultimaClave:= cota superior de las claves;
  while not EoF(f) and ultimaClave > c do begin
    Read(f, elem);
    if c = elem.clave then
      encontrado:= True;
    ultimaClave:= elem.clave
  end {while}
end; {BuscarOrd}
```

15.4 Mezcla y ordenación de archivos secuenciales

Una de las operaciones fundamentales en el proceso de archivos es el de su ordenación, de forma que las componentes contiguas cumplan una cierta relación de orden.

Cuando los archivos tengan un tamaño pequeño pueden leerse y almacenarse en un array, pudiéndose aplicar las técnicas de ordenación estudiadas para arrays. Estas técnicas son también aplicables a archivos de acceso directo (véase el apartado B.9).

Por último, cuando el único acceso permitido a los archivos es el secuencial, como en los archivos de Pascal, es necesario recurrir a algoritmos específicos para dicha ordenación.

El método más frecuente de ordenación de archivos secuenciales es en realidad una variante no recursiva del algoritmo *Merge Sort* estudiado para arrays en el apartado anterior. Consiste en dividir el archivo origen en dos archivos auxiliares y después mezclarlos ordenadamente sobre el propio archivo origen, obteniendo de esta forma, al menos, pares de valores ordenados. Si ahora se vuelve a dividir el archivo origen y se mezclan otra vez, se obtienen, al menos, cuádruplas de valores ordenados, y así sucesivamente hasta que todo el archivo esté ordenado.

A este proceso que usa dos archivos auxiliares se le denomina de *mezcla simple*. Si se usan más archivos auxiliares, se llama de *mezcla múltiple*, con el que se puede mejorar el tiempo de ejecución.

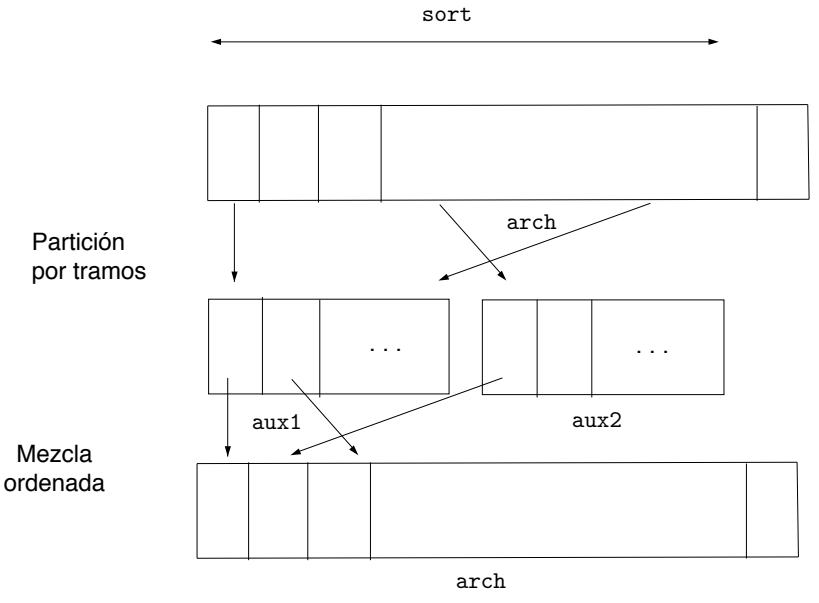


Figura 15.8.

Por lo tanto se tienen dos acciones diferentes: por una parte hay que dividir el archivo original en otros dos, y por otra hay que mezclar ordenadamente los dos archivos auxiliares sobre el archivo original, como se puede ver en la figura 15.8.

Vamos a concentrarnos primero en esta segunda acción de mezcla ordenada, ya que tiene entidad propia dentro del proceso de archivos: en el caso de disponer de dos archivos ordenados, situación que se da con relativa frecuencia, la mezcla ordenada garantiza que el archivo resultante también esté ordenado.

Para realizar la mezcla se ha de acceder a las componentes de los dos archivos, utilizando el operador ^, que aplicaremos a los archivos `aux1` y `aux2`. Si la primera componente de `aux1` es menor que la de `aux2`, se escribe en el archivo `arch` y se accede al siguiente valor de `aux1`. En caso contrario, se escribe en `arch` la primera componente de `aux2` avanzando en este archivo. En el caso de que se llegara, por ejemplo, al final del archivo `aux1`, hay que copiar las restantes componentes de `aux2` en `arch`. Si por el contrario, terminara `aux2`, habremos de copiar las restantes componentes de `aux1` en `arch`.

Dados los valores de `aux1` y `aux2`, la secuencia de lecturas y escrituras sería la que se muestra el al figura 15.9.

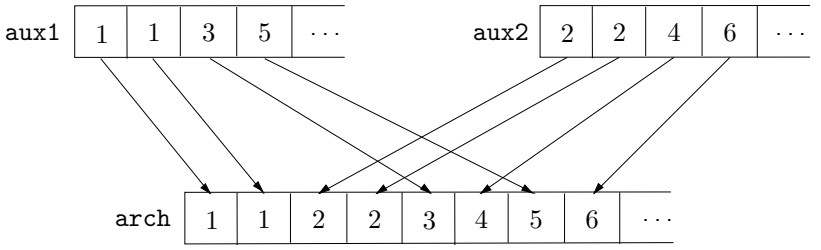


Figura 15.9.

Veamos el diseño descendente de `Mezcla`:

mientras no se acabe `aux1` *y no se acabe* `aux2` *hacer*
  si `aux1^ < aux2^` *entonces*
    `arch^:= aux1^`
    Avanzar `aux1`
    Poner en `arch`
  fin {*si*}
  si `aux2^ < aux1^` *entonces*
    `arch^:= aux2^`
    Avanzar `aux2`
    Poner en `arch`
  fin {*si*}
fin {*mientras*}
mientras no se acabe `aux1` *hacer*
  `arch^:= aux1^`
  Avanzar `aux1`
  Poner en `arch`
fin {*mientras*}
mientras no se acabe `aux2` *hacer*
  `arch^:= aux2^`
  Avanzar `aux2`
  Poner en `arch`
fin {*mientras*}

A continuación se ha escrito el procedimiento `Mezcla` en Pascal:

```
procedure Mezcla(var aux1, aux2, arch:  archivo);
  {Efecto:  arch := mezcla ordenada de aux1 y aux2}
begin
  Reset(aux1);
  Reset(aux2);
  ReWrite(arch);
```

```
    while not EoF(aux1) and not EoF(aux2) do
      if aux1^ < aux2^ then begin
        arch^:= aux1^;
        Put(arch);
        Get(aux1)
      end {if}
      else begin
        arch^:= aux2^;
        Put(arch);
        Get(aux2)
      end; {else}
    while not EoF(aux1) do  begin
        {Se copia en arch el resto de aux1, si es necesario}
      arch^:= aux1^;
      Put(arch);
      Get(aux1)
    end; {while}
    while not EoF(aux2) do  begin
        {Se copia en arch el resto de aux2, en caso necesario}
      arch^:= aux2^;
      Put(arch);
      Get(aux2)
    end; {while}
    Close(arch);
    Close(aux1);
    Close(aux2)
  end;  {Mezcla}
```

Como puede apreciarse el procedimiento `Mezcla` que se propone utiliza el cursor de archivo, lo que no está permitido en Turbo Pascal (véase el apartado B.9). Para poder utilizar dicho procedimiento en Turbo Pascal hay que modificarlo, utilizando en su lugar el procedimiento `Mezcla` que se detalla seguidamente.

El nuevo procedimiento `Mezcla` dispone, a su vez, de dos procedimientos anidados: el primero, llamado `LeerElemDetectandoFin`, comprueba si se ha alcanzado el final del archivo, y en caso contrario lee una componente del archivo. El segundo, `PasarElemDetectando`, escribe una componente dada en el archivo de destino, y utiliza el procedimiento `LeerElemDetectandoFin` para obtener una nueva componente. A continuación se muestra el procedimiento `Mezcla` modificado:

```
  procedure Mezcla(var aux1, aux2, arch:  tArchivo);
    {Efecto:  arch := mezcla ordenada de aux1 y aux2}
    var
      c1,c2:  tComponente;
      finArch1, finArch2:  boolean;
```

```
  procedure LeerElemDetectandoFin(var arch:  tArchivo;
                var comp:  tComponente; var finArch:  boolean);
  begin
    finArch:= EoF(arch);
    if not finArch then
      Read(arch, comp)
  end; {LeerElemDetectandoFin}

  procedure PasarElemDetectandoFin(var archOrigen, archDestino:
      tArchivo; var comp:  tComponente; var finArchOrigen:  boolean);
  begin
    Write(archDestino, comp);
    LeerElemDetectandoFin(archOrigen, comp, finArchOrigen)
  end; {PasarElemDetectandoFin}

begin  {Mezcla}
  Reset(aux1);
  Reset(aux2);
  Rewrite(arch);
  LeerElemDetectandoFin(aux1, c1, finArch1);
  LeerElemDetectandoFin(aux2, c2, finArch2);
  while not finArch1 and not finArch2 do
    if c1 < c2 then
      PasarElemDetectandoFin (aux1, arch, c1, finArch1)
    else
      PasarElemDetectandoFin (aux2, arch, c2, finArch2);
  while not finArch1 do
    PasarElemDetectandoFin (aux1, arch, c1, finArch1);
  while not finArch2 do
    PasarElemDetectandoFin (aux2, arch, c2, finArch2);
  Close(arch);
  Close(aux1);
  Close(aux2)
end; {Mezcla}
```

Abordamos ahora el desarrollo de la otra acción a realizar en este algoritmo, que es la división de `arch` en los dos archivos auxiliares `aux1` y `aux2`. Para optimizar esta tarea conviene tener en cuenta los posibles tramos ordenados que existan ya en `arch` para no desordenarlos. Así, mientras los sucesivos valores que se van leyendo de `arch` estén ordenados, los vamos escribiendo, por ejemplo, en `aux1`. En el momento en que una componente de `arch` esté desordenada pasamos a escribir en `aux2` donde seguiremos escribiendo los sucesivos valores de `arch` que formen otro tramo ordenado. Al aparecer un nuevo valor fuera de orden cambiamos de nuevo a `aux2`.

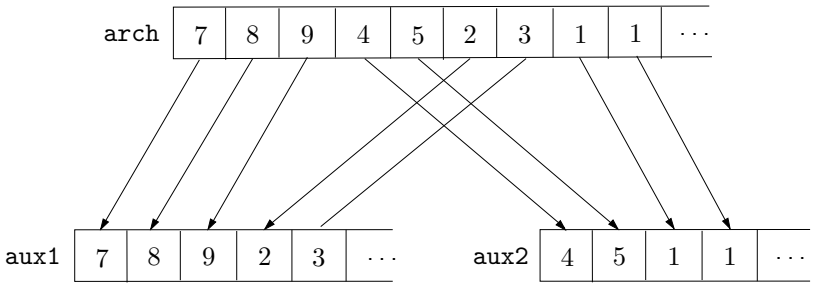


Figura 15.10.

Dados los valores de `arch`, un ejemplo de división sería la que aparece en la figura 15.10.

Este método de división hace que los archivos tengan el mismo número de tramos o a lo sumo que difieran en uno, por lo que su mezcla se denomina *mezcla equilibrada*. Además, se detecta inmediatamente si `arch` está ordenado, ya que existirá un único tramo que se escribirá en `aux1`, quedando `aux2` vacío.

Para saber cuándo cambiar de archivo auxiliar hay que comprobar si el `valorAnterior` leído es mayor que el `valorActual`, lo que obliga a realizar una lectura inicial, sin comparar, para tener un valor asignado a `valorAnterior`, y a almacenar en una variable booleana `cambio` el destino actual. Al escribir al menos una componente en `aux2`, éste ya no está vacío, lo que señalizaremos con una variable booleana `esVacio2`.

Veamos cómo podría ser un primer esbozo de `Division`:

```
si no se ha acabado arch entonces
  Leer valorActual de arch
  Escribir valorActual en aux1
  valorAnterior:= valorActual
fin {si}
mientras no se acabe arch hacer
  Leer valorActual de arch
  si valorAnterior > valorActual entonces
    Cambiar cambio
  si cambio entonces
    Escribir valorActual en aux1
  si no
    Escribir valorActual en aux2
    esVacio2:= False
  fin {si no}
  valorAnterior:= valorActual
fin {mientras}
```

A continuación se ha implementado el procedimiento `Division` en Pascal:

```
procedure Division(var arch, aux1, aux2:  tArchivo;
                   var esVacio2:  boolean);
   {Efecto:  arch se divide en dos archivos aux1 y aux2,
    copiando alternativamente tramos ordenados maximales}

   var
     valorActual, valorAnterior:  tComponente;
     cambio:  boolean;
       {conmuta la escritura en aux1 y aux2}

begin
   Reset(arch);
   ReWrite (aux1);
   ReWrite (aux2);
   cambio:= True;
   esVacio2:= True;
   if not EoF(arch) then begin
     Read(arch, valorActual);
     Write(aux1, valorActual);
     valorAnterior:= valorActual
   end;
   while not EoF(arch) do begin
       {se buscan y copian los tramos ordenados}
     Read(arch, valorActual);
     if valorAnterior > valorActual then
       cambio:= not cambio;
     if cambio then
       Write(aux1, valorActual)
     else begin
       Write(aux2, valorActual);
       esVacio2:= False
     end;
     valorAnterior:= valorActual
   end; {while}
   Close(arch);
   Close(aux1);
   Close(aux2)
end; {Division}
```

15.5 Ejercicios

1. Se desea examinar el funcionamiento del método de búsqueda por bipartición del siguiente modo:

```
Número buscado:  27
  1  4  5 12 25 27 31 42 43 56 73 76 78 80 99
  [                      <                    ]
  [        >         ]
              [  =  ]
El 27 está en la 6ª posición
```

   Modifique el programa dado en el apartado 15.1.3 para que dé su salida de esta forma.

2. Para examinar igualmente la evolución de los métodos de ordenación por intercambio, también se pueden insertar los siguientes subprogramas:

   (a) Uno de escritura que muestre el contenido de un vector (que, supuestamente, cabe en una línea) después de cada intercambio que se produzca.

   (b) Otro que sitúe las marcas `'>'` y `'<'` debajo de las componentes que se intercambian para facilitar su seguimiento visual.

   Desarrolle los subprogramas descritos e incorpórelos en un programa que muestre de este modo los intercambios que lleve a cabo.

3. **Ordenación por el método de la burbuja**

   Un método de ordenación consiste en examinar todos los pares de elementos contiguos (intercambiándolos si es preciso), efectuando este recorrido hasta que, en una pasada, no sea preciso ningún intercambio. Desarrolle un procedimiento para ordenar un vector siguiendo este algoritmo.

4. Desarrolle un procedimiento para ordenar una matriz de $m \times n$ considerando que el último elemento de cada fila y el primero de la siguiente son contiguos.

5. Dados dos archivos secuenciales ordenados ascendentemente, escriba un programa que los intercale, produciendo otro con los componentes de ambos ordenados ascendentemente.

6. **Ordenación por separación**

   Sea un archivo de texto compuesto por una línea de enteros separados por espacios.

   (a) Escriba un subprograma que construya otro archivo a partir del original aplicando la siguiente operación a cada línea:

   - Si la línea está vacía, se ignora.
   - Si la línea tiene un solo elemento, se copia.
   - De lo contrario, se extrae su primer elemento y se construyen tres líneas en el archivo nuevo: la primera formada por los elementos menores que el primero, en la segunda el primero, y en la tercera los mayores que el primero.

(b) Escriba un programa que repita el proceso anterior hasta obtener un archivo formado tan sólo por líneas unitarias, copiándolas entonces en una sola línea.

(c) Establezca las semejanzas que encuentre entre este algoritmo y uno de ordenación de vectores.

7. **Ordenación por mezcla**

Sea un archivo de texto compuesto por una línea de enteros separados por espacios.

(a) Escriba un subprograma que construye otro archivo, a partir del original, de la siguiente forma:

- Primero se separan los enteros, situando uno en cada línea.
- Luego se intercalan las líneas de dos en dos (véase el ejercicio 5). Este paso se repetirá hasta llegar a un archivo con una sola línea.

(b) Establezca las semejanzas que encuentre entre este algoritmo y uno de ordenación de vectores.

8. Complete el programa del ejercicio 6 con las siguientes opciones:

(a) Ordenar Inventario: que ordene el vector donde se almacenan los productos, utilizando alguno de los métodos de ordenación presentados en el capítulo, de las siguientes formas:

- De forma ascendente por su nombre, y en caso de que el nombre coincida, por su marca, y en caso de que ésta también coincidiera, de forma descendente por su tamaño.
- De forma descendente por el valor obtenido al multiplicar su precio por el número de unidades inventariadas.

(b) Ordenar Archivo: que ordene un archivo dado, donde se almacenan los productos, utilizando el método de División y Mezcla, en las formas señaladas en el apartado primero

15.6 Referencias bibliográficas

Con relación al contenido de esta parte del libro, en [Aho88] encontramos un tratamiento a un nivel elevado del tipo compuesto conjunto y de los algoritmos de búsqueda y ordenación; en particular, se recomienda la lectura del capítulo 8 donde se abordan los principales esquemas de clasificación interna, y el capítulo 11 donde se recogen los de clasificación externa.

Otro texto interesante es [Coll87], con una clara orientación a Pascal. Los capítulos de mayor interés son el 4 dedicado a los conjuntos, el 5 dedicado a la estructura vectorial, el 12 sobre ordenación de vectores y el 14 que trata sobre archivos. El texto presenta un enfoque moderno y utiliza especificaciones formales.

En esta bibliografía no podemos dejar de citar el texto clásico [Wir86], y en particular sus dos primeros temas sobre estructuras fundamentales de datos y ordenación de vectores y archivos. Utiliza técnicas de refinamientos progresivos, pero no las especificaciones formales.

Otro de los textos más utilizados sobre estos temas es [Dale89], que se caracteriza por la gran cantidad de ejemplos resueltos. Los capítulos 9 y del 11 al 15 son los que tratan sobre tipos de datos.

Un texto más reciente es [Sal93], que en su parte cuatro refleja las estructuras de datos. Hace una presentación progresiva de los contenidos con ejercicios de complejidad creciente. Faltan por concretar algunos métodos de ordenación de vectores y de archivos secuenciales.

El método de los vectores paralelos (apartado 15.2.6) no es nuevo; por ejemplo, se estudia en [DL89], aunque con el nombre de *punteros de ordenación*.

Tema V

Memoria dinámica

Tema V

Memoria dinámica

Capítulo 16

Punteros

Las estructuras de datos estudiadas hasta ahora se almacenan estáticamente en la memoria física del computador. Cuando se ejecuta un subprograma, se destina memoria para cada variable global del programa y tal espacio de memoria permanecerá reservado durante toda su ejecución, se usen o no tales variables; por ello, en este caso hablamos de asignación *estática* de memoria.

Esta rigidez presenta un primer inconveniente obvio: las estructuras de datos estáticas no pueden crecer o menguar durante la ejecución de un programa. Obsérvese sin embargo que ello no implica que la cantidad de memoria usada por un programa durante su funcionamiento sea constante, ya que depende, por ejemplo, de los subprogramas llamados. Y, más aún, en el caso de subprogramas recursivos se podría llegar fácilmente a desbordar la memoria del computador.

Por otra parte, la representación de ciertas construcciones (como las listas) usando las estructuras conocidas (concretamente los arrays) tiene que hacerse situando elementos consecutivos en componentes contiguas, de manera que las operaciones de inserción de un elemento nuevo o desaparición de uno ya existente requieren el desplazamiento de todos los posteriores para cubrir el vacío producido, o para abrir espacio para el nuevo.

Estos dos aspectos, tamaño y disposición rígidos, se superan con las llamadas estructuras de datos *dinámicas*. La definición y manipulación de estos objetos

se efectúa en Pascal mediante un mecanismo nuevo (el puntero), que permite al programador referirse directamente a la memoria.

Además estas estructuras de datos son más flexibles en cuanto a su forma: árboles de tamaños no acotados y con ramificaciones desiguales, redes (como las ferroviarias por ejemplo), etc.

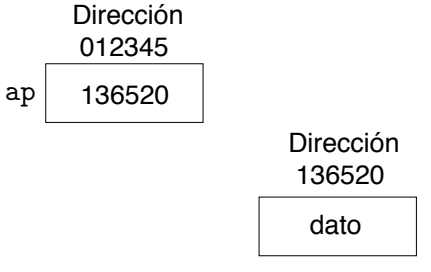
No obstante, esta nueva herramienta no está exenta de peligros en manos de un programador novel, que deberá tratar con mayor cuidado las estructuras creadas, atender especialmente al agotamiento del espacio disponible, etc.

En este capítulo se estudia el concepto de puntero o apuntador (en inglés *pointer*) así como sus operaciones asociadas. Éste es pues un capítulo técnico, que muy bien podría dar como primera impresión que las aportaciones de este nuevo mecanismo son sobre todo dificultades y peligros. Naturalmente, no es así: una vez identificados los peligros y superados los detalles técnicos (capítulo presente), el siguiente capítulo logrará sin duda convencer de que las múltiples ventajas superan con creces los inconvenientes mencionados.

16.1 Introducción al uso de punteros

Un *puntero* es una variable que sirve para señalar la posición de la memoria en que se encuentra otro dato almacenando como valor la dirección de ese dato.

Para evitar confusión entre la variable puntero y la variable a la que apunta (o variable *referida*) conviene imaginar gráficamente este mecanismo. En la siguiente figura se muestra la variable puntero `ap`, almacenada en la dirección 012345, y la celda de memoria que contiene la variable a la que apunta.



Puesto que *no* es el contenido real de la variable puntero lo que nos interesa, sino el de la celda cuya dirección contiene, es más común usar el siguiente diagrama

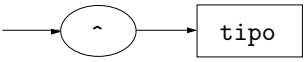
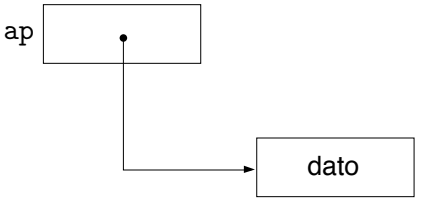


Figura 16.1. Definición de un tipo puntero.



que explica por sí mismo el porqué de llamar puntero a la variable `ap`.

Este último diagrama muestra que un puntero tiene dos componentes: la dirección de memoria a la que apunta (contenido del puntero) y el elemento referido (contenido de la celda de memoria cuya dirección está almacenada en el puntero).

Al usar punteros conviene tener muy claro que la variable puntero y la variable a la que apunta son dos variables distintas y, por lo tanto, sus valores son también distintos. A lo largo del capítulo se tendrá ocasión de diferenciar claramente entre ambas.

16.1.1 Definición y declaración de punteros

Antes de poder usar punteros en un programa es necesario declararlos: en primer lugar habrá que definir el tipo del dato apuntado y, posteriormente, declarar la(s) variable(s) de tipo puntero que se usará(n).

Por consiguiente, una variable puntero sólo puede señalar a objetos de un mismo tipo, establecido en la declaración. Así por ejemplo, un puntero podrá señalar a caracteres, otro a enteros y otro a vectores pero, una vez que se declara un puntero, sólo podremos usarlo para señalar variables del tipo para el cual ha sido definido. Esta exigencia permite al compilador mantener la consistencia del sistema de tipos, así como conocer la cantidad de memoria que debe reservar o liberar para el dato apuntado.

El tipo puntero es un tipo de datos simple. El diagrama sintáctico de su definición aparece en la figura 16.1, en la que hay que resaltar que el circunflejo (`^`) indica que se está declarando un puntero a variables del tipo `tipo`.

Naturalmente, para usar punteros en un programa, no basta con definir el tipo puntero: es necesario declarar alguna variable con este tipo. En este fragmento de programa se define el tipo `tApuntChar` como un puntero a variables de tipo `char` y, posteriormente, se declara la variable `apCar` de tipo puntero a caracteres.

```
type
  tApuntChar = ^char;
var
  apCar:  tApuntChar
```

Una variable de tipo puntero ocupa una cantidad de memoria fija, independientemente del tipo del dato señalado, ya que su valor es la dirección en que reside éste. Por otra parte, si bien el tipo de la variable `apCar` es `tApuntChar`, el dato señalado por `apCar` se denota mediante `apCar^`, cuyo tipo es por lo tanto `char`.

Como ocurre con las otras variables, el valor de un puntero estará en principio indefinido. Pero, además, los objetos señalados no tienen existencia inicial, esto es, ni siquiera existe un espacio en la memoria destinado a su almacenamiento. Por ello, no es correcto efectuar instrucciones como `WriteLn(apCar1^)` hasta que se haya creado el dato apuntado. Estas operaciones se estudian en el siguiente apartado.

16.1.2 Generación y destrucción de variables dinámicas

En la introducción del tema se destacó la utilidad de la memoria dinámica, de variables que se generan cuando se necesitan y se destruyen cuando han cumplido su cometido. La creación y destrucción de variables dinámicas se realiza por medio de los procedimientos predefinidos `New` y `Dispose`, respectivamente. Así pues, la instrucción

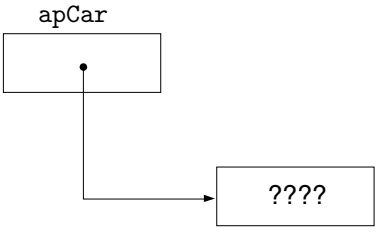
```
New(apCar)
```

tiene un efecto doble:

1. Reserva la memoria para un dato del tipo apropiado (en este caso del tipo `char`).

2. Coloca la dirección de esta nueva variable en el puntero.

que, gráficamente, se puede expresar así:




Obsérvese que la operación de generación `New` ha generado el dato apuntado por `apCar`, pero éste contiene por el momento una información desconocida.

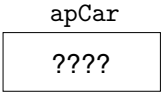
Para destruir una variable dinámica se usa el procedimiento estándar `Dispose`. La instrucción

```
Dispose(apCar)
```

realiza las dos siguientes acciones

1. Libera la memoria asociada a la variable referida `apCar^` (dejándola disponible para otros fines).
2. Deja indefinido el valor del puntero.

Gráficamente, esos efectos llevan a la siguiente situación:



En resumen, una variable dinámica sólo se creará cuando sea necesario (lo que ocasiona la correspondiente ocupación de memoria) y, previsiblemente, se destruirá una vez haya cumplido con su cometido (con la consiguiente liberación de la misma).

16.1.3 Operaciones básicas con datos apuntados

Recuérdese que el dato referido por el puntero `apCar` se denota `apCar^`, que es de tipo `char`. Por consiguiente, son válidas las instrucciones de asignación, lectura y escritura y demás operaciones legales para los caracteres:

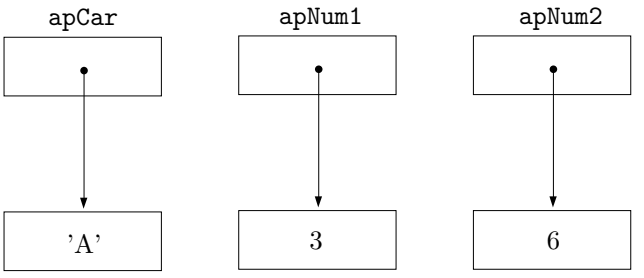


Figura 16.2.

```
type
  tApCaracter = ^char;
var
  apCar: tApCaracter;
...
New(apCar);
ReadLn(apCar^); {supongamos que se da la letra 'B'}
apCar^:= Pred(apCar^);
WriteLn(apCar^);
```

escribiéndose en el `output` la letra `'A'`.

Igualmente, suponiendo que se ha declarado

```
type
  tApNumero = ^integer;
var
  apNum1, apNum2: tApNumero;
```

el siguiente fragmento de programa es válido:

```
New(apNum1);
New(apNum2);
apNum1^:= 2;
apNum2^:= 4;
apNum2^:= apNum1^ + apNum2^;
apNum1^:= apNum2^ div 2;
```

y ambos fragmentos de instrucciones llevan a la situación de la figura 16.2, que será referida varias veces en este apartado.

En general, las operaciones válidas con un dato apuntado dependen de su tipo. Por ejemplo, el fragmento de programa siguiente

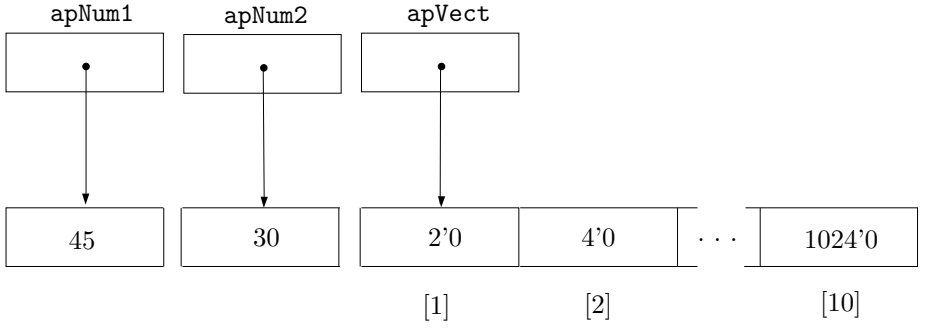


Figura 16.3.

```
type
  tVector10 = array[1..10] of real;
  tApNumero = ^integer;
  tApVector10 = ^tVector10;
var
  apNum1, apNum2: ApNumero;
  apVect: tApVector10;
  i: integer;
  ...
  New(apNum1);
  New(apNum2);
  New(apVect);
  apNum1^:= 45;
  apNum2^:= 30;
  apVect^[1]:= 2;
  for i:= 2 to 10 do
    apVect^[i]:= apVect^[i-1] * 2;
```

deja el estado de la memoria como se indica en la figura 16.3.

Por su parte, las operaciones siguientes, por ejemplo, no serían legales:

```
ReadLn(apVect^); {Lectura de un array de un solo golpe}
apNum1^:= apNum2^ + apVect^[1]; {Tipos incompatibles}
```

16.1.4 Operaciones básicas con punteros

Sólo las operaciones de comparación (con la igualdad) y asignación están permitidas entre punteros. En la situación de la figura 16.2, la comparación `apNum1 = apNum2` resulta ser falsa. Más aún, tras ejecutar las instrucciones

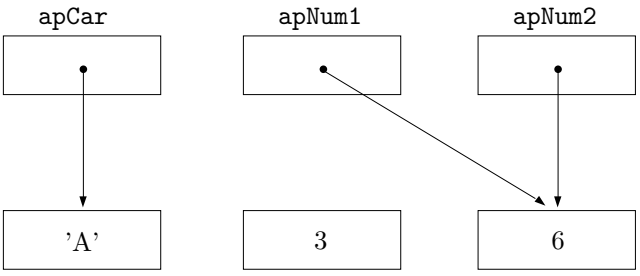


Figura 16.4.

```
apNum1^:= 6;
apNum2^:= 6;
```

la comparación:

```
apNum1 = apNum2
```

seguiría siendo `False`, ya que las direcciones apuntadas no coinciden, a pesar de ser iguales los datos contenidos en dichas direcciones.

También es posible la asignación

```
apNum1:= apNum2
```

cuyo resultado puede verse en la figura 16.4 en la que se observa que ambos punteros señalan a la misma dirección, resultando ahora iguales al compararlos:

```
apNum1 = apNum2
```

produce un resultado `True` y, como consecuencia, `apNum1^` y `apNum2^` tienen el mismo valor, 6. Además, esta coincidencia en la memoria hace que los cambios efectuados sobre `apNum1^` o sobre `apNum2^` sean indistintos:

```
ApNum1^:= 666;
WriteLn(ApNum2^);  {666}
```

Obsérvese que el espacio de memoria reservado inicialmente por el puntero `apNum1` sigue situado en la memoria. Lo adecuado en este caso habría sido liberar ese espacio con `Dispose` antes de efectuar esa asignación.

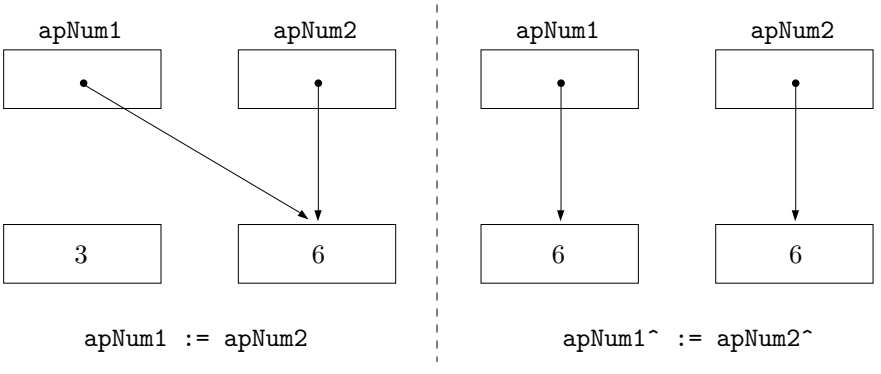
En ambos casos, asignación y comparación, es necesario conservar la consistencia de los tipos utilizados. Por ejemplo, en la situación representada en la figura 16.2, la siguiente asignación es errónea

```
apCar:= apNum1
```

porque, aunque ambas variables sean punteros, en su declaración queda claro que no apuntan a variables del mismo tipo.

Obsérvese que las instrucciones `apNum1:= apNum2` y `apNum1^:= apNum2^` tienen distinto efecto. A continuación se muestra gráficamente el resultado de cada una de las instrucciones anteriores, partiendo de la situación de la figura 16.2,

Supuesto que `apNum1^=3` y `apNum2^=6` sean `True`



donde se observa que, al asignar los punteros entre sí, se *comparte* la variable referida; en nuestro caso la celda de memoria que contenía a `apNum1^` se pierde, sin posibilidad de recuperación (a menos que otro puntero señalara a esta celda). Por otra parte, al asignar las variables referidas se tiene el mismo valor *en dos celdas distintas* y los punteros permanecen sin variar.

16.1.5 El valor *nil*

Un modo alternativo para dar valor a un puntero es, simplemente, diciendo que no apunta a ningún dato. Esto se puede conseguir utilizando la constante predefinida **nil**.

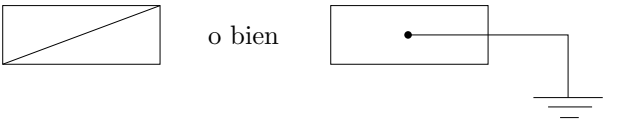


Figura 16.5. puntero iniciado en **nil**.

Por ejemplo, la siguiente asignación define el puntero `apCar`:

```
apCar:= nil
```

Gráficamente, el hecho de que una variable puntero no apunte a nada se representa cruzando su celda de memoria con una diagonal, o bien mediante el símbolo (prestado de la Electricidad) de conexión a tierra, como se indica en la figura 16.5.

En cuanto al tipo del valor **nil** es de resaltar que esta constante es común a cualquier tipo de puntero (sirve para todos), pues sólo indica que el puntero está anulado.

En el capítulo siguiente se pondrá de manifiesto la utilidad de esta operación. Por el momento, digamos solamente que es posible saber si un puntero está anulado, mediante la comparación con **nil**, por ejemplo mediante `apNum1 = nil`.

Obsérvese que no es necesario iniciar los punteros apuntando a **nil**. En el ejemplo anterior se inician usando `New`, esto es, creando la variable referida.

16.2 Aplicaciones no recursivas de los punteros

Se ha comentado en la introducción que las aplicaciones más importantes de los punteros, están relacionadas con las estructuras de datos recursivas, que estudiaremos en el capítulo siguiente. El apartado que pone cierre a éste presenta algunas situaciones con estructuras no recursivas en las que los punteros resultan útiles.

El hecho de que los punteros sean objetos de tipo simple es interesante y permite, entre otras cosas:

1. La asignación de objetos no simples en un solo paso.

2. La definición de funciones cuyo resultado no es simple.

16.2.1 Asignación de objetos no simples

Esta aplicación adquiere mayor relevancia cuando se consideran registros de gran tamaño en los que el problema es el elevado coste al ser necesaria la copia de todas sus componentes. Por ejemplo, en la situación

```
type
  tFicha = record
    nombre:  ...
    direccion:  ...
    ...
  end; {tFicha}
var
  pers1, pers2: tFicha;
```

la asignación `pers1:= pers2` es altamente costosa si el tipo `tFicha` tiene grandes dimensiones.

Una forma de economizar ese gasto consiste en usar sólo su posición, haciendo uso de punteros:

```
var
  p1, p2:  ^tFicha;
```

en vez de las variables `pers1` y `pers2`. Entonces, la instrucción

```
p1:= p2
```

es casi instantánea, por consistir tan sólo en la copia de una dirección.

El artificio anterior resulta útil, por ejemplo, cuando hace falta intercambiar variables de gran tamaño. Si se define el tipo

```
type
  tApFicha = ^tFicha;
```

el procedimiento siguiente efectúa el intercambio rápidamente, con independencia del tamaño del tipo de datos `fichas`:

```
procedure Intercambiar(var p1, p2: tApFicha);
  var
    aux: tApFicha;
begin
  aux:= p1;
  p1:= p2;
  p2:= aux
end; {Intercambiar}
```

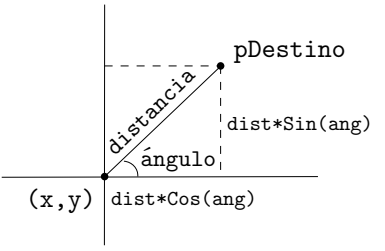


Figura 16.6.

La ordenación de vectores de elementos grandes es un problema en el que la aplicación mostrada demuestra su utilidad. Por ejemplo, la definición

```
type
  tListaAlumnos = array [1..100] of tFicha;
```

se puede sustituir por un array de punteros,

```
type
  tListaAlumnos = array [1..100] of tApFicha;
```

de este modo la ordenación se realizará de una manera mucho más rápida, debido esencialmente al tiempo que se ahorra al no tener que copiar literalmente todas las fichas que se cambian de posición.

16.2.2 Funciones de resultado no simple

Esta aplicación tiene la misma base que la anterior, cambiar el objeto por el puntero al mismo. A continuación vemos un sencillo ejemplo en el que se aprovecha esta característica.

Supongamos, por ejemplo, que hemos de definir un programa que, dado un punto del plano, un ángulo y una distancia, calcule un nuevo punto, alcanzado al recorrer la distancia según el rumbo dado, según la figura 16.6.

Una primera aproximación al diseño del programa podría ser la siguiente:

Lectura de datos: punto base, distancia y ángulo
Cálculo del punto final
Escritura de resultados

Naturalmente, en primer lugar se deben definir los tipos, constantes y variables que se usarán en el programa: se definirá un tipo `tPunto` como un registro

de dos componentes de tipo real y un tipo dinámico `tApPunto` que señalará al tipo `punto`, además se necesitarán dos variables `distancia` y `angulo` para leer los valores de la distancia y el ángulo del salto, una de tipo `punto` para el origen del salto y otra de tipo `tApPunto` para el punto de destino.

```
type
  tPunto = record
    x, y: real
  end; {tPunto}
  tApPunto = ^tPunto;
var
  angulo, distancia: real;
  origen: tPunto;
  pDestino: tApPunto;
```

El cálculo del punto final se puede realizar mediante la definición (y posterior aplicación) de una función. Ésta tendría que devolver un punto, formado por dos coordenadas; dado que esto no es posible, se devolverá un puntero al tipo `tPunto` (de ahí la necesidad del tipo `tApPunto`). Dentro de la función utilizaremos el puntero `pPun` para generar la variable apuntada y realizar los cálculos, asignándolos finalmente a `Destino` para devolver el resultado de la función. La definición de la función es absolutamente directa:

```
function Destino(orig: tPunto; ang, dist: real): tApPunto;
  var
    pPun:  tApPunto;
begin
  New(pPun);
  pPun^.x:= orig.x + dist * Cos(ang);
  pPun^.y:= orig.y + dist * Sin(ang);
  Destino:= pPun
end; {Destino}
```

Finalmente, si se añaden los procedimientos de lectura de datos y escritura de resultados tenemos el siguiente programa:

```
Program Salto (input, output);
  type
    tPunto = record
      x, y: real
    end {tPunto};
    tApPunto = ^tPunto;
```

```
  var
    angulo, distancia: real;
    origen: tPunto;
    pDestino: tApPunto;


  function Destino(orig: tPunto; ang, dist: real): tApPunto;
    var
      pPun:  tApPunto;
  begin
    New(pPun);
    pPun^.x:= orig.x + dist * Cos(ang);
    pPun^.y:= orig.y + dist * Sin(ang)
    Destino:= pPun
  end; {Destino}

begin
  Write('Introduzca la x y la y del punto origen: ');
  ReadLn(origen.x, origen.y);
  Write('Introduzca el rumbo y la distancia: ');
  ReadLn(angulo, distancia);
  pDestino:= Destino(origen, angulo, distancia);
  WriteLn('El punto de destino es:');
  WriteLn('X = ', pDestino^.x:20:10,' Y = ', pDestino^.y:20:10)
end. {Salto}
```

La idea más importante de este programa estriba en que la función devuelve un valor de tipo puntero, que es un tipo simple, y por lo tanto correcto, como resultado de una función. Sin embargo, la variable referenciada por el puntero es estructurada, y almacena las dos coordenadas del punto de destino. Mediante esta estratagema, conseguimos obtener un resultado estructurado de una función.

16.3 Ejercicios

1. Implementar algún algoritmo de ordenación de registros (por ejemplo, fichas de alumnos), mediante un array de punteros, tal como se propone en el apartado 16.2.1.

2. Complete los tipos de datos apropiados para que sea correcta la siguiente instrucción:

```
for i:= 1 to n do begin
  New(p);
  p^:= i
end {for}
```

(a) ¿Cuál es el efecto de ejecutarla?

(b) ¿Cuál es el efecto de añadir las siguientes instrucciones tras la anterior?

```
WriteLn(p);
WriteLn(p^);
Dispose(p)
```

(c) ¿De qué forma se pueden recuperar los valores $1...N-1$ generados en el bucle **for**?

3. Escriba una función que reciba una matriz cuadrada de 3×3 y calcule (si es posible) su inversa, devolviendo un puntero a dicha matriz.

Capítulo 17

Estructuras de datos recursivas

En este capítulo se introducen, a un nivel elemental, las estructuras de datos recursivas como la aplicación más importante de la memoria dinámica.

Se presentan las *listas*, el tipo de datos recursivo más sencillo y que tiene múltiples aplicaciones, y los casos particulares de las *pilas* y las *colas*, como listas con un "acceso controlado". También se introducen los árboles binarios como ejemplo de estructura de datos no lineal, y se termina apuntando hacia otras estructuras cuyo estudio queda fuera de nuestras pretensiones.

17.1 Estructuras recursivas lineales: las listas enlazadas

Las representaciones para listas que introducimos en este capítulo nos permitirán insertar y borrar elementos más fácil y eficientemente que en una implementación estática (para listas acotadas) usando el tipo de datos **array**.

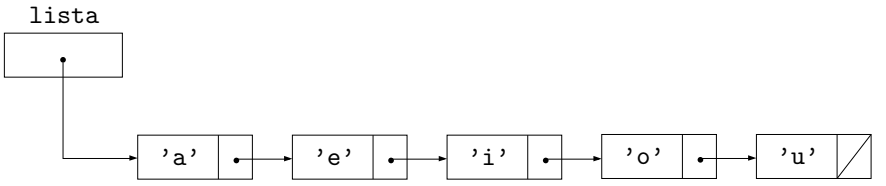


Figura 17.1. Representación de una lista enlazada dinámica.

A medida que desarrollemos las operaciones de inserción y borrado, centraremos nuestra atención en las acciones necesarias para mantener diferentes clases particulares de listas: las pilas y las colas. Por lo común, son las distintas operaciones requeridas las que determinarán la representación más apropiada para una lista.

17.1.1 Una definición del tipo lista

Una *lista* es una colección lineal de elementos que se llaman *nodos*. El término colección lineal debe entenderse de la siguiente manera: tenemos un primer y un último nodo, de tal manera que a cada nodo, salvo el último, le corresponde un único sucesor, y a cada nodo, salvo el primero, le corresponde un único predecesor. Se trata, pues, de una estructura de datos cuyos elementos están situados secuencialmente.

Una definición del tipo listas se puede realizar usando punteros y registros. Para ello consideramos que cada nodo de la lista es un registro con dos componentes: la primera almacena el `contenido` del nodo de la lista y, la segunda, un puntero que señala al `siguiente` elemento de la lista, si éste existe, o con el valor **nil** en caso de ser el último. Esta construcción de las listas recibe el nombre de *lista enlazada dinámica*. En la figura 17.1 se muestra gráficamente esta idea.

Esencialmente, una lista será representada como un puntero que señala al principio (o cabeza) de la lista. La definición del tipo `tLista` de elementos de tipo `tElem` se presenta a continuación, junto con la declaración de una variable del tipo lista:

```
type
  tElem = char;  {o lo que corresponda}
  tLista = ^tNodo;
  tNodo = record
    contenido: tElem;
    siguiente: tLista
  end; {tNodo}
```

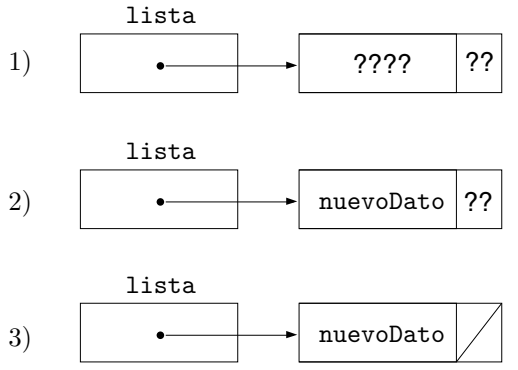


Figura 17.2. Inserción de un elemento en una lista vacía.

```
var
  lista : tLista;
```

Sobre el código anterior debe señalarse que se define `tLista` como un puntero a un tipo no definido todavía. Esto está permitido en Pascal (supuesto que tal tipo es definido posteriormente, pero en el mismo grupo de definiciones) precisamente para poder construir estructuras recursivas.

17.1.2 Inserción de elementos

Supóngase que nuestra `lista` está iniciada con el valor **nil**. Para introducir un elemento `nuevoDato` en ella, habrá que completar la siguiente secuencia de pasos, que se muestra gráficamente en la figura 17.2:

1. En primer lugar, habrá que generar una variable del tipo `tNodo`, que ha de contener el nuevo eslabón: esto se hace mediante la sentencia `New(lista)`.

2. Posteriormente, se asigna `nuevoDato` al campo `contenido` del nodo recién generado. La forma de esta asignación dependerá del tipo de datos de la variable `nuevoDato`.

3. Y por último, hay que anular (con el valor **nil**) el campo `siguiente` del nodo para indicar que es el último de la `lista`.

Para insertar un `nuevoDato` al principio de una lista no vacía, `lista`, se ha de proceder como se indica (véase la figura 17.3):

1. Una variable auxiliar `listaAux` se usa para apuntar al principio de la `lista` con el fin de no perderla. Esto es:

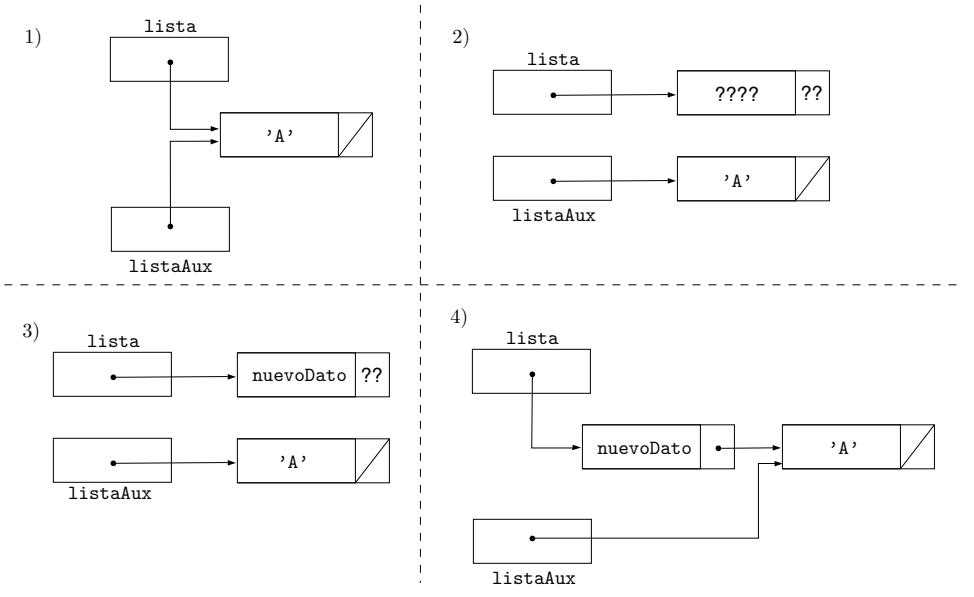


Figura 17.3. Adición de un elemento al principio de una lista.

```
listaAux:= lista
```

2. Después se asigna memoria a una variable del tipo `tNodo` (que ha de contener el nuevo elemento): esto se hace mediante la sentencia `New(lista)`.

3. Posteriormente, se asigna `nuevoDato` al campo `contenido` del nuevo nodo:

```
listaˆ.contenido:= nuevoDato
```

4. Y, por último, se asigna la `listaAux` al campo `siguiente` del nuevo nodo para indicar los demás elementos de la lista, es decir:

```
listaˆ.siguiente:= listaAux
```

A continuación se van a definir algunas de las operaciones relativas a listas. Para empezar, se desarrolla un procedimiento para añadir un elemento al principio de una lista, atendiendo al diseño descrito por los pasos anteriores:

```
procedure AnnadirPrimero(var lista: tLista; nuevoDato: tElem);
  {Efecto:  se añade nuevoDato al comienzo de lista}
  var
    listaAux: tLista;
begin
  listaAux:= lista;
  New(lista);
  lista^.contenido:= nuevoDato;
  lista^.siguiente:= listaAux
end; {AnnadirPrimero}
```

Obsérvese que este procedimiento sirve tanto para cuando `lista` es vacía como para cuando no lo es.

17.1.3 Eliminación de elementos

A continuación se presenta el procedimiento `Eliminar` que sirve para eliminar el primer elemento de una lista no vacía. La idea es bastante simple: sólo hay que actualizar el puntero para que señale al siguiente elemento de la lista (si existe).

```
procedure EliminarPrimero(var lista: tLista);
  {PreC.:  la lista no está vacía}
  {Efecto:  el primer nodo ha sido eliminado}
  var
    listaAux: tLista;
begin
  listaAux:= lista;
  lista:= lista^.siguiente;
  Dispose(listaAux)
end; {EliminarPrimero}
```

17.1.4 Algunas funciones recursivas

La longitud de una lista

La naturaleza recursiva de la definición dada del tipo lista permite definir fácilmente funciones que trabajen usando un proceso recursivo. Por ejemplo, la función `Longitud` de una lista se puede definir recursivamente (es evidente que el caso base es `Longitud(nil) = 0`) siguiendo la línea del siguiente ejemplo:

```
Longitud ([a,b,c])  ⇝  1 + Longitud ([b,c])
                    ⇝  1 + (1 + Longitud([c]))
                    ⇝  1 + (1 + (1 + Longitud(nil)))
                    ⇝  1 + (1 + (1 + 0)) ⇝ 3
```

De aquí se deduce que, si `lista` no está vacía, y llamamos `Resto(lista)` a la `lista` sin su primer elemento, tenemos:

```
Longitud (lista) = 1 + Longitud (Resto(lista))
```

Por lo tanto, en resumen,

$$\texttt{Longitud(lista)} = \begin{cases} \texttt{0} & \text{si } \texttt{lista = nil} \\ \texttt{1 + Longitud (Resto(lista))} & \text{en otro caso} \end{cases}$$

Finalmente, `Resto(lista)` se representa en Pascal como `listaˆ.siguiente`. Por lo tanto, la definición recursiva en Pascal de la función `Longitud` es la siguiente:

```
type
  natural = 0..MaxInt;
...
function Longitud(lista: tLista): natural;
  {Dev.  el número de nodos de lista}
begin
  if lista = nil then
    Longitud:= 0
  else
    Longitud:= 1 + Longitud(listaˆ.siguiente)
end; {Longitud}
```

El uso de la recursividad subyacente en la definición del tipo de datos lista hace que la definición de esta función sea muy simple y clara, aunque también es posible y sencilla una versión iterativa, que se deja como ejercicio.

Evaluación de polinomios: La regla de Horner

Supongamos que se plantea escribir un programa para evaluar en un punto dado x un polinomio con coeficientes reales

$$P(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \cdots + a_{n-1}x + a_n,$$

Conocido `x`, un posible modo de evaluar el polinomio es el siguiente:

Conocido `x`
Dar valor inicial 0 a `valor`
  para cada `i` *entre 0 y* `N`
    Añadir $\texttt{a}_{\texttt{N-i}}$ `*` $\texttt{x}^{\texttt{i}}$ `a valor`
  Devolver `valor`

La *regla de Horner* consiste en disponer el polinomio de forma que su evaluación se pueda hacer simplemente mediante la repetición de operaciones aritméticas elementales (y no tener que ir calculando los valores de las potencias sucesivas de `x`).

No es difícil observar que

$$P(x) = \Big(\cdots \Big(((a_0 x + a_1)x + a_2)x + a_3 \Big) x + \cdots a_{n-1} \Big) x + a_n$$

y que, con esta expresión, el valor del polinomio se puede calcular de acuerdo con el siguiente diseño:

Dar valor inicial $\texttt{a}_0$ *a* `valor`
para cada `i` *entre 1 y* `N` *hacer*
  `valor:=` $\texttt{a}_\texttt{i}$`+ x * valor`
Devolver `valor`

En este apartado volvemos a considerar la representación de polinomios en un computador una vez que se han introducido algunas herramientas de programación dinámica.

Hay dos formas de representar un polinomio conocida la lista de sus coeficientes: de más significativo a menos, o viceversa. A saber:

$$\begin{aligned} P_1(x) &= a_0 x^n + a_1 x^{n-1} + a_2 x^{n-2} + \cdots + a_{n-1} x + a_n \\ &= \Big(\cdots \Big(((a_0 x + a_1)x + a_2)x + a_3 \Big) x + \cdots a_{n-1} \Big) x + a_n \\ P_2(x) &= b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1} + b_n x^n \\ &= b_0 + x \Big(b_1 + x \Big(b_2 + x(b_3 + \cdots + x(b_{n-1} + x b_n) \cdots) \Big) \Big) \end{aligned}$$

En el supuesto de que tengamos los coeficientes ordenados de grado menor a mayor (si lo están del otro modo la modificación del programa es mínima) según su grado y almacenados en una lista (`lCoef`), es posible dar una versión recursiva de la función anterior.

La recursividad de la regla de Horner se aprecia mejor si consideramos la siguiente tabla, en la que se renombra el resultado de cada uno de los paréntesis de la expresión de Horner:

$$\begin{aligned} c_0 &= a_0 \\ c_1 &= a_1 + x c_0 \end{aligned}$$

$$\begin{aligned} c_2 &= a_2 + xc_1 \\ c_3 &= a_3 + xc_2 \\ \vdots &= \quad \vdots \\ c_{n-1} &= a_{n-1} + xc_{n-2} \\ c_n &= a_n + xc_{n-1} \end{aligned}$$

Claramente se observa que $b_n = P(x)$ y, en consecuencia, ya podemos escribir su expresión recursiva.

```
Horner(x,lCoef) =
        lCoef^.contenido + x * Horner(x,lCoef^.siguiente)
```

El caso base se da cuando la lista de coeficientes se queda vacía (**nil**). Un sencillo ejemplo nos muestra que, en este caso, la función debe valer cero.

Supongamos que queremos evaluar un polinomio constante $P(x) = K$, entonces su lista de coeficientes es `[K]`, usando la expresión recursiva anterior tendríamos que

```
Horner(x,nil) = 0
Horner(x,[K]) = K + x * Horner(x,nil)
```

Una vez conocidos el caso base y la expresión recursiva definiremos una función `Horner2` con dos variables: el punto donde evaluar y una lista de reales que representan los coeficientes del polinomio ordenados de mayor a menor grado.

```
function Horner2(x: real; lCoef: tLista): real;
  {Dev.  P(x), siendo P el polinomio de coeficientes lCoef}
begin
  if lCoef = nil then
    Horner2:= 0
  else
    Horner2:= lCoef^.contenido + x * Horner2(x,lCoef^.siguiente)
end; {Horner2}
```

17.1.5 Otras operaciones sobre listas

A continuación se enumeran algunas de las operaciones que más frecuentemente se utilizan al trabajar con listas:

1. Determinar el número de elementos de una lista.

2. Leer o modificar el k-ésimo elemento de la lista.
3. Insertar o eliminar un elemento en la k-ésima posición.
4. Insertar o eliminar un elemento en una lista ordenada.
5. Combinar dos o más listas en una única lista.
6. Dividir una lista en dos o más listas.
7. Ordenar los elementos de una lista de acuerdo con un criterio dado.
8. Insertar o eliminar un elemento en la k-ésima posición de una lista de acuerdo con un criterio dado.
9. Buscar si aparece un valor dado en algún lugar de una lista.

Como ejemplo de implementación de algunas de estas operaciones, a continuación se presentan procedimientos que eliminan o insertan un nuevo elemento en una lista.

El procedimiento `EliminarK`

En primer lugar, se presenta un procedimiento que elimina el `k`-ésimo elemento de una lista (que se supone de longitud mayor que `k`). En el caso en que `k` = 1, podemos utilizar el procedimiento `EliminarPrimero` desarrollado anteriormente, por lo que en este apartado nos restringimos al caso `k` > 1.

El primer esbozo de este procedimiento es muy sencillo:

Localizar el nodo `(k-1)`*-ésimo de* `lista`
Asociar el puntero `siguiente del nodo (k-1)`*-ésimo al nodo* `(k+1)`*-ésimo*

El proceso se describe gráficamente en la figura 17.4.

Emplearemos un método iterativo para alcanzar el (`k` − 1)-ésimo nodo de la lista, de modo que se irá avanzando nodo a nodo desde la cabeza (el primer nodo de la lista) hasta alcanzar el (`k` − 1)-ésimo usando un puntero auxiliar `apAux`.

Para alcanzar el nodo (`k`−1)-ésimo, se empezará en el primer nodo, mediante la instrucción

```
listaAux:= lista;
```

y luego se avanzará iterativamente al segundo, tercero, ..., (`k` − 1)-ésimo ejecutando

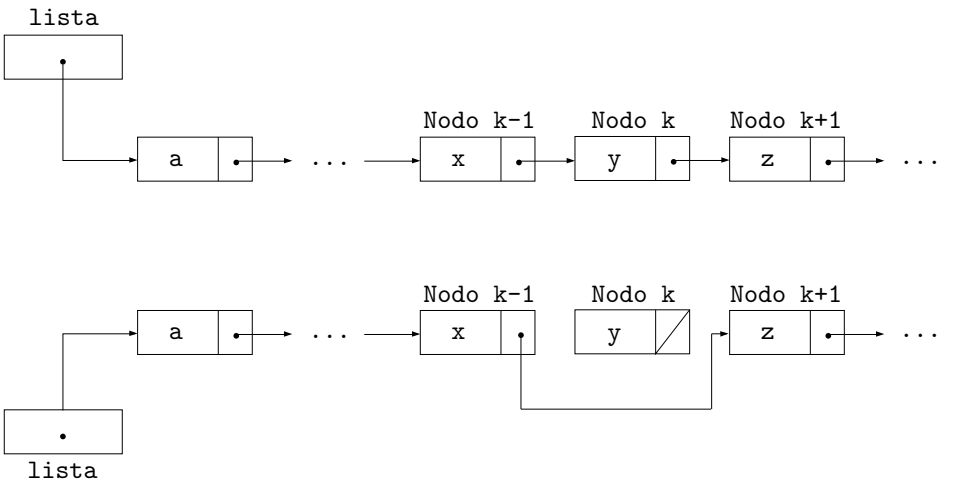


Figura 17.4. Eliminación del k-ésimo nodo de una lista.

```
for i:= 2 to k - 1 do
  apAux:= apAux^.siguiente
```

Una vez hecho esto, sólo hace falta saltarse el nodo k-ésimo, liberando después la memoria que ocupa. Con esto, el código en Pascal de este procedimiento podría ser:

```
procedure EliminarK(k: integer; var lista: tLista);
  {PreC.:  1 < k ≤ longitud de la lista}
  {Efecto:  se elimina el elemento k-ésimo de lista}
  var
    apAux, nodoSupr: tLista;
    i: integer;
begin
  apAux:= lista;
    {El bucle avanza apAux hasta el nodo k-1}
  for i:= 2 to k-1 do
    apAux:= apAux^.siguiente;
      {Actualizar punteros y liberar memoria}
    nodoSupr:= apAux^.siguiente;
    apAux^.siguiente:= nodoSupr^.siguiente;
    Dispose(nodoSupr)
  end; {EliminarK}
```

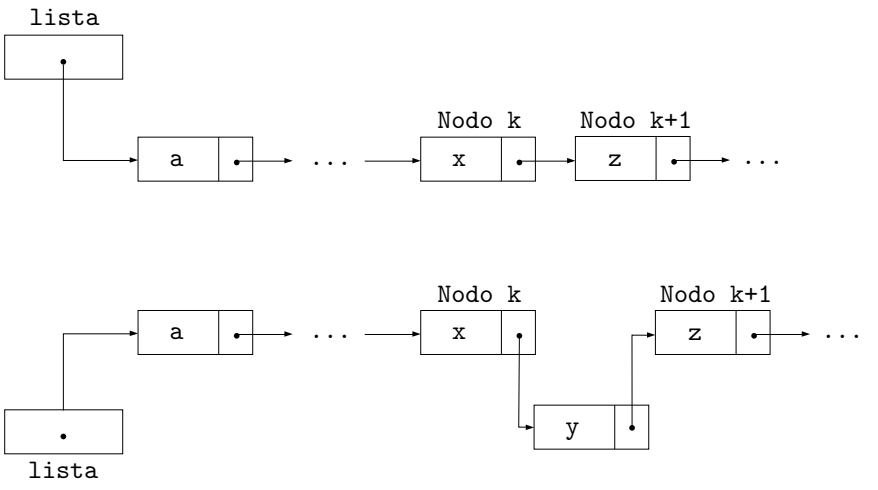


Figura 17.5. Inserción de un nodo tras el k-ésimo nodo de una lista.

Obsérvese que el procedimiento anterior funciona también cuando el nodo eliminado es el último nodo de la lista.

El procedimiento `InsertarK`

Otro procedimiento importante es el de inserción de un elemento tras cierta posición de la lista. En particular, a continuación se implementará la inserción de un nuevo nodo justo después del `k`-ésimo nodo de una lista (de longitud mayor que `k`). Puesto que la inserción de un nuevo nodo al comienzo de una lista ya se ha presentado, nos restringiremos al caso $\texttt{k} \geq 1$.

De nuevo el primer nivel de diseño es fácil:

Localizar el nodo `k`*-ésimo de* `lista`
Crear un `nuevoNodo` *y asignarle el contenido* `nuevoDato`
Asociar el puntero `siguiente` *del nodo* `k`*-ésimo a* `nuevoNodo`
Asociar el puntero `siguiente` *de* `nuevoNodo` *al nodo* `(k+1)`*-ésimo*

En la figura 17.5 se pueden observar las reasignaciones de punteros que hay que realizar. También en este caso es necesario usar un puntero auxiliar para localizar el nodo tras el cual se ha de insertar el nuevo dato y se volverá a hacer uso del bucle **for**.

La creación del nuevo nodo y la reasignación de punteros es bastante directa, una vez que `apAux` señala al nodo `k`-ésimo. El código correspondiente en Pascal podría ser:

```
New(nuevoNodo);
nuevoNodo^.contenido:= nuevoDato;
nuevoNodo^.siguiente:= apAux^.siguiente;
apAux^.siguiente:= nuevoNodo
```

Uniendo el bucle, para alcanzar el k-ésimo nodo, con las asignaciones anteriores se obtiene la implementación completa del procedimiento `InsertarK`:

```
procedure InsertarK(k: integer; nuevoDato: tElem; var lista: tLista);
  {PreC.:  k ≥ 1 y lista tiene al menos k nodos}
  {Efecto:  nuevoDato se inserta tras el k-ésimo nodo de lista}
  var
    nuevoNodo, apAux: tLista;
    i: integer;
begin
  apAux:= lista;
    {El bucle avanza apAux hasta el nodo k}
  for i:= 1 to k-1 do
    apAux:= apAux^.siguiente;
      {Actualización de punteros}
  New(nuevoNodo);
  nuevoNodo^.contenido:= nuevoDato;
  nuevoNodo^.siguiente:= apAux^.siguiente;
  apAux^.siguiente:= nuevoNodo
end; {InsertarK}
```

Son muy frecuentes las situaciones en las que se accede a una lista sólo a través de su primer o último elemento. Por esta razón tienen particular interés las implementaciones de listas en las que el acceso está restringido de esta forma, siendo las más importantes las pilas y las colas. A continuación se definen estos tipos de listas junto con determinadas operaciones asociadas: se dará su representación y las operaciones de modificación y acceso básicas, y se presentarán ejemplos donde se vea el uso de las mismas.

17.2 Pilas

Una *pila* es un tipo de lista en el que todas las inserciones y eliminaciones de elementos se realizan por el mismo extremo de la lista.

El nombre de pila procede de la similitud en el manejo de esta estructura de datos y la de una "pila de objetos". Estas estructuras también son llamadas listas LIFO,[1] acrónimo que refleja la característica más importante de las pilas.

[1]Del inglés *Last-In-First-Out*, es decir, el último en entrar es el primero en salir.

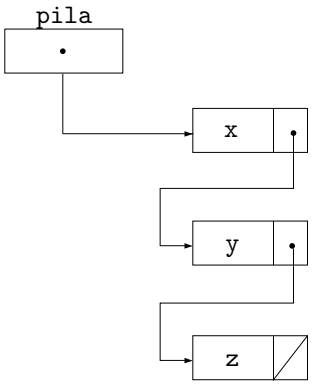


Figura 17.6. Representación gráfica del concepto de pila.

Es fácil encontrar ejemplos de pilas en la vida real: hay una pila en el aparato dispensador de platos de un autoservicio, o en el cargador de una pistola automática, o bien en el tráfico en una calle sin salida. En todos estos casos, el último ítem que se añade a la pila es el primero en salir. Gráficamente, podemos observar esto en la figura 17.6.

Dentro del contexto informático, una aplicación importante de las pilas se encuentra en la implementación de la recursión, que estudiaremos con cierto detalle más adelante.

17.2.1 Definición de una pila como lista enlazada

Puesto que una pila es una lista en la cual sólo se insertan o eliminan elementos por uno de sus extremos, podríamos usar la declaración dada de las listas y usarlas restringiendo su acceso del modo descrito anteriormente.

```
type
  tElem = char;  {o lo que corresponda}
  tPila = ^tNodoPila;
  tNodoPila = record
    contenido:  tElem;
    siguiente:  tPila
  end; {tNodoPila}
```

17.2.2 Operaciones básicas sobre las pilas

Entre las operaciones básicas del tipo pila se encuentran la creación de una pila, la consulta del contenido del primer elemento de la pila, la inserción de

un nuevo elemento sobre la pila (que se suele llamar `push`), y la eliminación del elemento superior de la pila (también llamada `pop`).[2]

Creación de una pila vacía

El primer procedimiento que consideramos es el de crear una pila vacía. La idea es bien simple: sólo hay que asignar el valor **nil** a su puntero.

```
procedure CrearPila(var pila: tPila);
  {PostC.:  pila es una pila vacía}
begin
  pila:= nil
end; {CrearPila}
```

Averiguar si una pila está vacía

Se trata sencillamente de la siguiente función

```
function EsPilaVacia(pila: tPila): boolean;
begin
  EsPilaVacia:= pila = nil
end; {EsPilaVacia}
```

Consulta de la cima de una pila

Una función especialmente importante es la que permite consultar el contenido del primer nodo de la pila. En esta implementación sólo hay que leer el campo `contenido` del primer nodo de la pila. La función tiene como argumento una variable de tipo `tPila` y como rango el tipo de sus elementos `tElem`. Está implementada a continuación bajo el nombre de `Cima`.

```
function Cima(pila: tPila): tElem;
  {PreC.:  pila no está vacía}
  {Dev.  el contenido del primer nodo de pila}
begin
  Cima:= pila^.contenido
end; {Cima}
```

[2]Los nombres de estas funciones proceden de los verbos usados en inglés para colocar o coger objetos apilados.

Añadir un nuevo elemento en la cima de la pila

Este procedimiento toma como parámetros una pila y la variable que se va a apilar; la acción que realiza es la de insertar el objeto como un nuevo nodo al principio de la pila. La definición de este procedimiento es directa a partir del método de inserción de un nodo al principio de una lista visto en el apartado anterior.

```
procedure Apilar(nuevoDato: tElem; var pila: tPila);
  {Efecto:  nuevoDato se añade sobre pila}
  var
    pilaAux: tPila;
begin
  pilaAux:= pila;
  New(pila);
  pila^.contenido:= nuevoDato;
  pila^.siguiente:= pilaAux
end; {Apilar}
```

Eliminar la cima de la pila

Para quitar un elemento de la pila sólo debemos actualizar el puntero. La definición del procedimiento no necesita mayor explicación, ya que es idéntica a la del procedimiento `Eliminar` presentado en el apartado 17.1.3.

```
procedure SuprimirDePila(var pila: tPila);
  {PreC.:  pila no está vacía}
  {Efecto:  se suprime el dato de la cima de la pila}
  var
    pilaAux:  tPila;
begin
  pilaAux:= pila;
  pila:= pila^.siguiente;
  Dispose(pilaAux)
end; {SuprimirDePila}
```

17.2.3 Aplicaciones

Pilas y recursión

Una aplicación importante de las pilas surge al tratar la recursión (véase el capítulo 10). En efecto, en cada llamada recursiva se añade una *tabla de activación* en una pila (denominada *pila recursiva*). Dicha tabla incorpora los argumentos y objetos locales con su valor en el momento de producirse la llamada.

Dicho de otro modo, la recursión se puede transformar en un par de bucles que se obtienen apilando las llamadas recursivas (primer bucle) para después ir evaluándolas una a una (con el segundo bucle).

Un ejemplo servirá para aclarar estas ideas; supongamos que tenemos una función definida recursivamente, como, por ejemplo, la función factorial (véase el apartado 10.1):

```
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}
```

Consideremos cómo se ejecuta la función `Fac` aplicada a un entero, por ejemplo el 3:

$$\begin{aligned} \texttt{Fac}(3) &= 3 * \texttt{Fac}(2) = 3 * 2 * \texttt{Fac}(1) \\ &= 3 * 2 * 1 * \texttt{Fac}(0) = 3 * 2 * 1 * 1 = 3! \end{aligned}$$

El primer bucle de los comentados más arriba consistiría en ir apilando los argumentos sucesivos de `Fac` hasta llegar al caso base, en este ejemplo tenemos `3, 2, 1`; el segundo bucle es el encargado de completar las llamadas recursivas usando la parte recursiva de `Fac`, esto es, se parte del caso base `Fac(0) = 1` y se van multiplicando los distintos valores apilados. Atendiendo a esta descripción, un primer nivel de diseño para la versión iterativa del factorial podría ser el siguiente:

```
para cada n entre num y  1
  Apilar n en pilaRec
Dar valor inicial 1 a fac
mientras que pilaRec no esté vacía hacer
  fac:= Cima(pilaRec) * fac
  SuprimirDePila(pilaRec)
Devolver fac
```

La implementación iterativa en Pascal de la función factorial se ofrece a continuación:

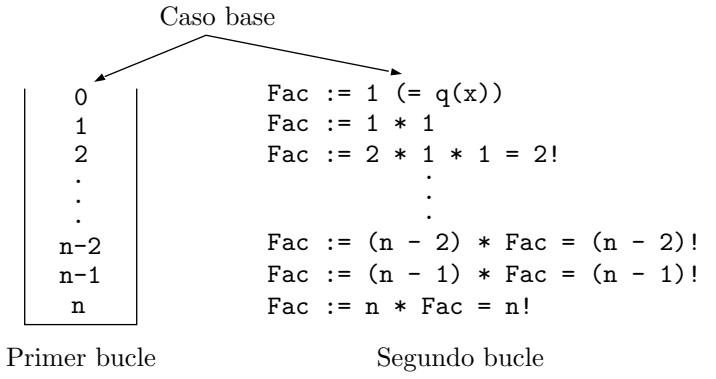


Figura 17.7. Expresión en dos bucles de la función factorial.

```
function FacIter (num: integer): integer;
  {PreC.:  num ≥ 0}
  {Dev.  num!}
  var
    pilaRec:  tPila;  {de enteros}
    n, fac:  integer;
begin
  n:= num;
  CrearPila(pilaRec);
    {Primer bucle:  acumulación de las llamadas}
  for n:= num downto 1 do
    Apilar (n, pilaRec);
      {Segundo bucle:  resolución de las llamadas}
  fac:= 1;  {Caso base}
  while pilaRec <> nil do begin
    fac:= Cima(pilaRec) * fac;
    SuprimirDePila(pilaRec)
  end; {while}
  FacIter:= fac
end; {FacIter}
```

En la figura 17.7 puede verse gráficamente el significado de los dos bucles que aparecen en el programa anterior.

La descomposición anterior de la función factorial no es más que un ejemplo del caso general de transformación de recursión en iteración que se expone seguidamente.

Supongamos que se tiene una función definida recursivamente, de la siguiente manera:

```
function F(x: tdato): tResultado;
begin
  if P(x) then
    F:= Q(x)
  else
    F:= E(x, F(T(x)))
end;  {F}
```

donde `P(x)` es una expresión booleana para determinar el caso base (en el factorial es `x = 0`, `Q(x)` es el valor de la función en el caso base, `T` es una función que transforma el argumento `x` en el de la siguiente llamada (en el factorial es `T(x) = x - 1`), y, finalmente, `E(x, y)` es la expresión que combina el argumento `x` con el resultado `y` devuelto por la llamada recursiva subsiguiente (en el factorial se tiene `E(x, y) = x * y`).

En resumen, cualquier función de la forma de la función `F` puede ser descrita mediante un par de bucles de forma que:

1. El primero almacena los parámetros de las sucesivas llamadas recursivas al aplicar la función `T` hasta llegar al caso base.

2. El segundo deshace la recursión aplicando la expresión `E(x, F(T(x))` repetidamente desde el caso base hasta el argumento inicial.

La descripción general de estos dos bucles se muestra a continuación:

```
function F(x: tDato): tResultado;
  var
    pilaRec: tPila;
    acum: tDato;
begin
  CrearPila(pilaRec);
    {Primer bucle}
  while not P(x) do begin
    Apilar(x, pilaRec);
    x:= T(x)
  end; {while}
  acum:= Q(x);  {Aplicación de F al caso base}
    {Segundo bucle}
  while pilaRec <> nil do begin
    acum:= E(Cima(pilaRec), acum);
    SuprimirDePila(pilaRec)
  end; {while}
  F:= acum
end; {F}
```

El proceso descrito anteriormente para una función recursiva puede llevarse a cabo también para procedimientos recursivos. En el apartado de ejercicios se propone su generalización a un procedimiento recursivo.

Evaluación postfija de expresiones

¿Quién no ha perdido algún punto por olvidarse un paréntesis realizando un examen? Este hecho, sin duda, le habrá llevado a la siguiente pregunta: ¿No será posible eliminar estos molestos signos del ámbito de las expresiones aritméticas? La respuesta es afirmativa: existe una notación para escribir expresiones aritméticas sin usar paréntesis, esta notación recibe el nombre de *notación postfija* o *notación polaca inversa*.

El adjetivo postfija o inversa se debe a que, en contraposición a la notación habitual (o infija) los operadores se ponen detrás de (y no entre) los argumentos. Por ejemplo, en lugar de escribir $a + b$ se ha de escribir $a\ b\ +$, y en vez de $a\ *\ (b\ +\ c)$ se escribirá $a\ b\ c\ +\ *$.

Para entender una expresión postfija hay que leerla de derecha a izquierda; la expresión anterior, por ejemplo, es un producto, si seguimos leyendo encontramos un +, lo cual indica que uno de los factores del producto es una suma. Siguiendo con la lectura vemos que tras el signo + aparecen c y b, luego el primer factor es $b + c$; finalmente hallamos a, de donde la expresión infija del miembro de la izquierda es $a * (b + c)$.

Aunque es probable que el lector no se encuentre a gusto con esta notación y comience a añorar el uso de paréntesis, la ventaja de la notación postfija reside en que es fácilmente implementable en un computador,[3] al no hacer uso de paréntesis ni de reglas de precedencia (esto es, convenios tales como que $a * b + c$ representa $(a * b) + c$ y no $a * (b + c)$).

La figura 17.8 muestra gráficamente el proceso de evaluación de la expresión $2\ 4\ *\ 3\ +$. Más detalladamente, haciendo uso de una pila, un computador interpretará esa expresión de la siguiente forma:

1. Al leer los dos primeros símbolos, 2 y 4, el computador aún no sabe qué ha de hacer con ellos, así que los pone en una pila.

2. Al leer el siguiente símbolo, $*$, saca dos elementos de la pila, los multiplica, y pone en ella su valor, 8.

3. El siguiente símbolo, 3, se coloca en la pila, pues no tenemos ningún operador que aplicar.

[3]De hecho, los compiladores de lenguajes de alto nivel traducen las expresiones aritméticas a notación postfija para realizar las operaciones. Hay lenguajes que usan siempre notación postfija, como, por ejemplo, el lenguaje de descripción de página PostScript.

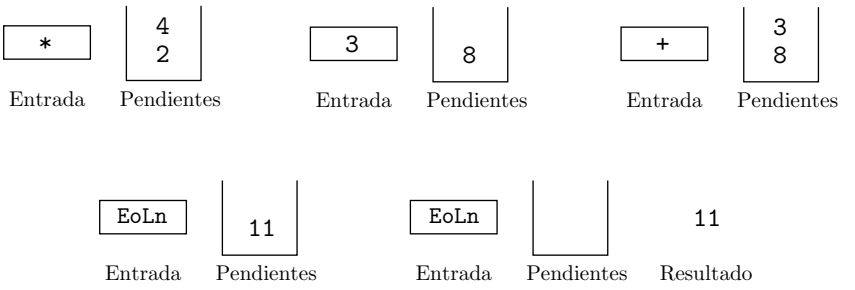


Figura 17.8. Evaluación postfija de una expresión.

4. Después se lee el símbolo +, con lo que se toman dos elementos de la pila, 3 y 8, se calcula su suma, 11, y se pone en la pila.

5. Para terminar, ya no hay ningún símbolo que leer y sólo queda el resultado de la operación.

Una vez visto el ejemplo anterior, la evaluación postfija de expresiones aritméticas es muy sencilla de codificar: sólo se necesita una pila para almacenar los operandos y las operaciones básicas de las pilas. La codificación en Pascal se deja como ejercicio.

17.3 Colas

Una *cola* es una lista en la que todas las inserciones se realizan por un extremo y todas las eliminaciones se realizan por el otro extremo de la lista.

El ejemplo más importante de esta estructura de datos, del cual recibe el nombre, es el de una cola de personas ante una ventanilla. En esta situación, el primero en llegar es el primero en ser servido; por esto, las colas también se llaman listas FIFO.[4]

Un ejemplo más interesante, dentro de un contexto informático, resulta al considerar la gestión de trabajos de una impresora conectada en red. Todos los archivos por imprimir se van guardando en una cola y se irán imprimiendo según el orden de llegada (sería ciertamente una ruindad implementar esta lista como una pila).

[4]Del inglés *First-In-First-Out*, es decir, el primero en entrar es el primero en salir.

17.3.1 Definición del tipo cola

Las colas, como listas que son, podrían definirse de la manera habitual, esto es:

```
type
  tElem = char;  {o lo que corresponda}
  tCola = ^tNodoCola;
  tNodoCola = record
    contenido:  tElem;
    siguiente:  tCola
  end; {tNodoCola}
```

Sin embargo, algunas operaciones, como poner un elemento en una cola, no resultan eficientes, debido a que debe recorrerse la lista en su totalidad para llegar desde su primer elemento hasta el último. Por ello, suele usarse otra definición, considerando una cola como un par de punteros:

```
type
  tElem = char;  {o lo que corresponda}
  tApNodo = ^tNodoCola;
  tNodoCola = record
    contenido: tElem;
    siguiente: tApNodo
  end; {tNodoCola}
  tCola = record
    principio: tApNodo;
    final: tApNodo
  end; {tCola}
```

Con esta definición de colas, cualquier operación que altere la posición de los nodos extremos de la lista deberá actualizar los punteros `principio` y `final`; sin embargo, esto resulta ventajoso en comparación con la obligatoriedad de recorrer toda la cola para añadir un nuevo elemento.

17.3.2 Operaciones básicas

Veamos cómo implementar las operaciones básicas con colas siguiendo la definición anterior.

Creación de una cola vacía

Para crear una cola se necesita iniciar a **nil** los campos `principio` y `final` del registro. Nada más fácil:

```
procedure CrearCola(var cola: tCola);
begin
  cola.principio:= nil;
  cola.final:= nil
end; {CrearCola}
```

Por eficiencia en la implementación de los siguientes procedimientos resultará conveniente considerar una lista vacía a aquella cuyo puntero `final` tiene el valor **nil**; esto permitirá que para comprobar si una cola es o no vacía sólo se necesite evaluar la expresión lógica `cola.final = nil`. Es trivial escribir el subprograma que efectúa esta comprobación.

Consulta del primer elemento

Esta operación es idéntica a `Cima` de una pila, puesto que nos interesa el `contenido` del primer nodo de la cola. Para verificar el nivel de comprensión sobre el tema, es un buen ejercicio intentar escribirla (sin mirar el análogo para pilas, como es natural).

Añadir un elemento

En las colas, los elementos nuevos se sitúan al final y por esta razón es necesario actualizar el puntero `final` (y también `principio` si la cola está vacía). Un simple gráfico servirá para convencerse cómo. El diseño de este procedimiento es el siguiente:

si `cola` *no está vacía entonces*
  Crear y dar valores al `nuevoNodo`
  Actualizar los punteros y el `final` *de* `cola`
en otro caso
  Crear y dar valores al `nuevoNodo`
  Actualizar `principio` *y* `final` *de* `cola`

La codificación en Pascal de este diseño podría ser la siguiente:

```
procedure PonerEnCola(dato: tElem; var cola: tCola);
  {Efecto:  dato se añade al final de cola}
  var
    nuevoNodo: tApNodo;
begin
  New(nuevoNodo);
  nuevoNodo^.contenido:= dato;
  nuevoNodo^.siguiente:= nil;
```

```
  if cola.final <> nil then begin
      {Si la cola no está vacía se actualizan los punteros}
    cola.final^.siguiente:= nuevoNodo;
    cola.final:= nuevoNodo
  end {then}
  else begin
      {Actualización de punteros:}
    cola.principio:= nuevoNodo;
    cola.final:= nuevoNodo
  end {else}
end; {PonerEnCola}
```

Suprimir el primer elemento

Para definir `SacarDeCola` tenemos que considerar dos casos distintos:

1. Que la cola sea unitaria, pues al sacar su único elemento se queda vacía, con lo que hay que actualizar tanto su `principio` como su `final`.

2. Que la cola tenga longitud mayor o igual a dos, en cuyo caso sólo hay que actualizar el campo `principio`.

A continuación se muestra la implementación del procedimiento:

```
procedure SacarDeCola(var cola: tCola);
  {PreC.:  cola es no vacía}
  {Efecto.:  se extrae el primer elemento de cola}
  var
    apuntaAux : tApNodo;
begin
  if cola.principio = cola.final then begin
      {La cola es unitaria}
    apuntaAux:= cola.principio;
    Dispose(apuntaAux);
    cola.principio:= nil;
    cola.final:= nil
  end {then}
  else begin
      {Si Longitud(cola) >= 2:}
    apuntaAux:= cola.principio;
    cola.principio:= apuntaAux^.siguiente;
    Dispose(apuntaAux)
  end {else}
end; {SacarDeCola}
```

17.3.3 Aplicación: gestión de la caja de un supermercado

Con la ayuda del tipo cola se presenta a continuación un programa de simulación del flujo de clientes en una caja de supermercado.

Una cantidad prefijada de clientes se va incorporando a la cola en instantes discretos con una cierta probabilidad también fijada de antemano;[5] cada cliente lleva un máximo de artículos, de forma que cada artículo se contabiliza en una unidad de tiempo. Cuando a un cliente le han sido contabilizados todos los artículos, es eliminado de la cola. En cada instante se muestra el estado de la cola.

Un primer esbozo en seudocódigo de este programa sería el siguiente:

Leer valores iniciales
Iniciar contadores
repetir
  Mostrar la cola
  Procesar cliente en caja
  Procesar cliente en cola
hasta que se acaben los clientes y los artículos

La lectura de los valores iniciales consiste en pedir al usuario del programa el número de clientes, `numClientes`, la probabilidad de que aparezca un cliente por unidad de tiempo, `probLlegada`, y el número máximo de artículos por cliente, `maxArti`. Tras definir los valores iniciales de los contadores de tiempo `t`, y de clientes puestos en cola, `contClientes`, sólo hay que refinar las acciones del bucle del diseño anterior. Así, en un nivel de diseño más detallado *Procesar cliente en caja* se descompondría en

si el cliente ya no tiene artículos entonces
  Retirarlo de la cola
en otro caso
  Reducir el número de artículos del cliente

Mientras que el refinamiento de *Procesar cliente en cola* sería

si quedan clientes y `random <` *probabilidad prefijada entonces*
  Añadir un cliente e incrementar el contador de clientes

A partir del refinamiento anterior ya podemos pasar a la implementación en Pascal de este programa.

[5] Para incluir factores de aleatoriedad se hace uso de `Randomize` y `Random` de Turbo Pascal (véase el apartado A.2.1).

El único tipo que se necesita definir para este programa es el tipo `tCola`. Las variables que almacenarán los datos del programa son el número de personas que llegarán a la caja, `numClientes`, el número máximo de artículos que cada persona puede llevar, `maxArti`, y la probabilidad de que una persona se una a la cola, `probLlegada`. Además se usará la variable `t` para medir los distintos instantes de tiempo, la variable `contClientes` para contar el número de personas que se han puesto en la cola y la variable `caja`, que es de tipo `tCola`.

El encabezamiento del programa es el siguiente:

```
Program SimuladorDeColas (input, output);
  type
    tElem = integer;
    tApNodo = ^tNodo;
    tNodo = record
      contenido: tElem;
      siguiente: tApNodo
    end; {tNodo}
    tCola = record
      principio, final: tApNodo
    end; {tCola}
  var
    t, contClientes, numClientes, maxArti: integer;
    probLlegada: real;
    caja: tCola;
```

Se usarán los siguientes procedimientos estándar del tipo `tCola`: `CrearCola`, `PonerEnCola`, `SacarDeCola`, y además `MostrarCola` que, como su nombre indica, muestra todos los elementos de la cola. Su implementación se presenta a continuación:

```
procedure MostrarCola(cola: tCola);
  {Efecto:  muestra en pantalla todos los elementos de cola}
  var
    apuntaAux: tApNodo;
begin
  if cola.final= nil then
    WriteLn('La caja está desocupada')
  else begin
    apuntaAux:= cola.principio;
    repeat
      WriteLn(apuntaAux^.contenido);
      apuntaAux:= apuntaAux^.siguiente
    until apuntaAux= nil
  end {else}
end; {MostrarCola}
```

En el cuerpo del programa, tal como se especifica en el diseño, tras la lectura de datos, se realiza la siguiente simulación en cada instante `t`: se procesa un artículo en la caja (se anota su precio), cuando se acaban todos los artículos de un determinado cliente éste se elimina de la cola (paga y se va); por otra parte se comprueba si ha llegado alguien (si su número aleatorio es menor o igual que `probLlegada`), si ha llegado se le pone en la cola y se actualiza el contador de personas.

Finalmente, el cuerpo del programa es el siguiente:

```
begin
  Randomize;
  CrearCola(caja);
  t:= 0;
  contClientes:= 0;
  WriteLn('Simulador de Colas');
  WriteLn('Introduzca número de personas');
  ReadLn(numClientes);
  WriteLn('Introduzca la probabilidad de llegada');
  ReadLn(probLlegada);
  WriteLn('Introduzca máximo de artículos');
  ReadLn(maxArti);
  repeat
    Writeln('Tiempo t =',t);
    MostrarCola(caja);
    t:= t + 1;
    if caja.principio^.contenido= 0 then
      SacarDeCola(caja)
    else with caja.principio^ do
      contenido:= contenido - 1;
    if (Random <= probLlegada) and (contClientes < numClientes)
    then begin
      PonerEnCola (Random(maxArti) + 1, caja);
      contClientes:= contClientes + 1
    end; {if}
  until (contClientes = numClientes) and (caja.principio= nil)
end. {SimuladorDeColas}
```

17.4 Árboles binarios

Es posible representar estructuras de datos más complejas que las listas haciendo uso de los punteros. Las listas se han definido como registros que contienen datos y un puntero al siguiente nodo de la lista; una generalización del concepto de lista es el árbol, donde se permite que cada registro del tipo de dato dinámico tenga más de un enlace. La naturaleza lineal de las listas hace

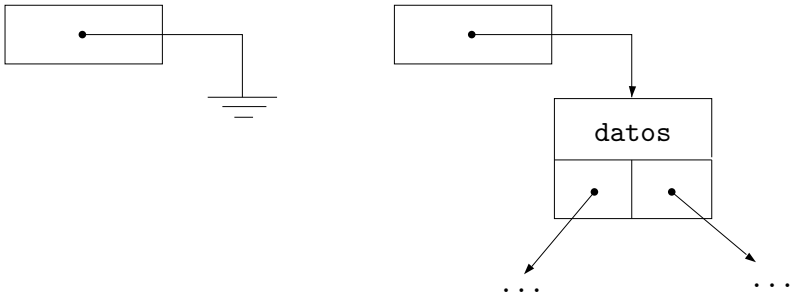


Figura 17.9.

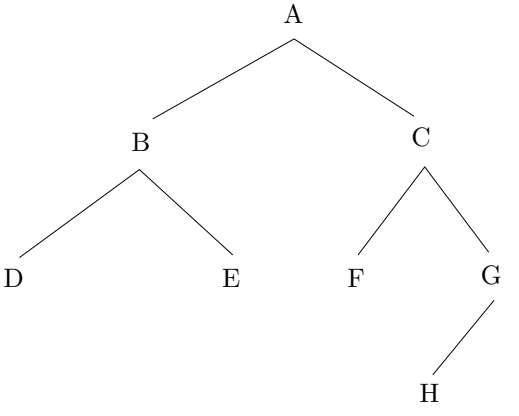


Figura 17.10.

posible, y fácil, definir algunas operaciones de modo iterativo; esto no ocurre con los árboles, que son manejados de forma natural haciendo uso de la recursión.

Los árboles son estructuras de datos recursivas más generales que una lista y son apropiados para aplicaciones que involucran algún tipo de jerarquía (tales como los miembros de una familia o los trabajadores de una organización), o de ramificación (como los árboles de juegos), o de clasificación y/o búsqueda. La definición recursiva de árbol es muy sencilla: Un *árbol* o es vacío o consiste en un nodo que contiene datos y punteros hacia otros árboles. Es decir, la representación gráfica de un árbol es una de las dos que aparecen en la figura 17.9.

En este apartado sólo trataremos con *árboles binarios*, que son árboles en los que cada nodo tiene a lo sumo dos descendientes. En la figura 17.10 vemos la representación gráfica de un árbol binario.

La terminología usada cuando se trabaja con árboles es, cuando menos, curiosa. En ella se mezclan conceptos botánicos como raíz y hoja y conceptos genealógicos tales como hijo, ascendientes, descendientes, hermanos, padres, etc. En el árbol de la figura 17.10, el nodo A es la *raíz* del árbol, mientras que los nodos D, E, F y H son las *hojas* del árbol; por otra parte, los *hijos* de la raíz son los nodos B y C, el *padre* de E es B, ...

La definición en Pascal del tipo árbol binario es sencilla: cada nodo del árbol va a tener dos punteros en lugar de uno.

```
type
  tElem = char;   {o el tipo que corresponda}
  tArbol = ^tNodoArbol;
  tNodoArbol = record
    hIzdo, hDcho: tArbol;
    contenido: tElem
  end; {tNodoArbol}
```

La creación de un árbol vacío, a estas alturas, no representa problema alguno. Sin embargo no ocurre lo mismo para realizar una consulta, ya que debemos saber cómo movernos dentro del árbol.

17.4.1 Recorrido de un árbol binario

Definir un algoritmo de recorrido de un árbol binario no es una tarea directa ya que, al no ser una estructura lineal, existen distintas formas de recorrerlo. En particular, al llegar a un nodo podemos realizar una de las tres operaciones siguientes:

(i) Leer el valor del nodo.

(ii) Seguir por el hijo izquierdo.

(iii) Seguir por el hijo derecho.

El orden en el que se efectúen las tres operaciones anteriores determinará el orden en el que los valores de los nodos del árbol son leídos. Si se postula que siempre se leerá antes el hijo izquierdo que el derecho, entonces existen tres formas distintas de recorrer un árbol:

Preorden: Primero se lee el valor del nodo y después se recorren los subárboles. Esta forma de recorrer el árbol también recibe el nombre de *recorrido primero en profundidad*.

El árbol de la figura 17.10 recorrido en preorden se leería así: ABDECFGH.

Inorden: En este tipo de recorrido, primero se recorre el subárbol izquierdo, luego se lee el valor del nodo y, finalmente, se recorre el subárbol derecho.

El árbol de la figura 17.10 recorrido en inorden se leería así: DBEAFCHG.

Postorden: En este caso, se visitan primero los subárboles izquierdo y derecho y después se lee el valor del nodo.

El árbol de la figura 17.10 recorrido en postorden se leería así: DEBFHGCA.

Ahora, escribir un procedimiento recursivo para recorrer el árbol (en cualquiera de los tres órdenes recién definidos) es tarea fácil, por ejemplo:

```
procedure RecorrerEnPreorden(arbol: tArbol);
begin
  if arbol <> nil then begin
    Visitarnodo(arbol); {por ejemplo, Write(arbol^.contenido)}
    RecorrerEnPreorden(arbol^.hIzdo);
    RecorrerEnPreorden(arbol^.hDcho);
  end {if}
end; {RecorrerEnPreorden}
```

para utilizar otro orden en el recorrido sólo hay que cambiar el orden de las acciones *Visitarnodo* y las llamadas recursivas.

17.4.2 Árboles de búsqueda

Como un caso particular de árbol binario se encuentran los *árboles binarios de búsqueda* (o *árboles de búsqueda binaria*), que son aquellos árboles en los que el valor de cualquier nodo es mayor que el valor de su hijo izquierdo y menor que el de su hijo derecho. Según la definición dada, no puede haber dos nodos con el mismo valor en este tipo de árbol.

Obsérvese que los nodos de un árbol binario de búsqueda se pueden enumerar en orden creciente siguiendo un recorrido en inorden.

La utilidad de los árboles binarios de búsqueda reside en que si buscamos cierta componente, podemos decir en qué mitad del árbol se encuentra comparando solamente con el nodo raíz. Nótese la similitud con el método de búsqueda binaria.

Una mejora de los árboles de búsqueda consiste en añadir un campo *clave* en cada nodo y realizar las búsquedas comparando los valores de dichas claves en lugar de los valores del campo `contenido`. De esta forma, pueden existir en el árbol dos nodos con el mismo valor en el campo contenido pero con clave distinta. En este texto se implementan los árboles de búsqueda sin campo clave para simplificar la presentación; la modificación de la implementación para incluir un campo clave es un ejercicio trivial.

Operaciones básicas

Las operaciones básicas para el manejo de árboles de búsqueda son la consulta, la inserción y la eliminación de nodos. Las dos primeras son de fácil implementación, haciendo uso de la natural recursividad de los árboles. La operación de eliminación de nodos es, sin embargo, algo más compleja, como se detalla a continuación.

Búsqueda de un nodo

Debido al orden intrínseco de un árbol de búsqueda binaria, es fácil implementar una función que busque un determinado valor entre los nodos del árbol y, en caso de encontrarlo, proporcione un puntero a ese nodo. La versión recursiva de la función es particularmente sencilla, todo consiste en partir de la raíz y rastrear el árbol en busca del nodo en cuestión, según el siguiente diseño:

si `arbol` *es vacío entonces*
  Devolver fallo
en otro caso si `arbol^.contenido = dato` *entonces*
  Devolver el puntero a la raíz de `arbol`
en otro caso si `arbol^.contenido > dato` *entonces*
  Buscar en el hijo izquierdo de `arbol`
en otro caso si `arbol^.contenido < dato` *entonces*
  Buscar en el hijo derecho de `arbol`

La codificación en Pascal es directa:

```
function Encontrar(dato: tElem; arbol: tArbol): tArbol;
  {Dev.  un puntero al nodo con dato, si dato está en arbol, o
  nil en otro caso}
begin
  if arbol = nil then
    Encontrar:= nil
  else with arbol^ do
    if dato < contenido then
```

```
      Encontrar:= Encontrar (dato, hIzdo)
    else if datos > contenido then
      Encontrar:= Encontrar (dato, hDcho)
    else Encontrar:= arbol
end; {Encontrar}
```

Inserción de un nuevo nodo

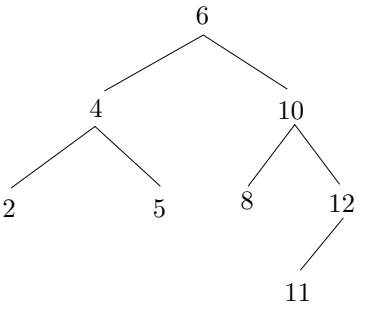
El siguiente procedimiento inserta un nuevo nodo en un árbol binario de búsqueda `árbol`; la inserción del nodo es tarea fácil, todo consiste en encontrar el lugar adecuado donde insertar el nuevo nodo, esto se hace en función de su valor de manera similar a lo visto en el ejemplo anterior. El seudocódigo para este procedimiento es:

si `arbol` *es vacío entonces*
  crear nuevo nodo
en otro caso si `arbol^.contenido > datoNuevo` *entonces*
  Insertar ordenadamente en el hijo izquierdo de `arbol`
en otro caso si `arbol^.contenido < datoNuevo` *entonces*
  Insertar ordenadamente en el hijo derecho de `arbol`

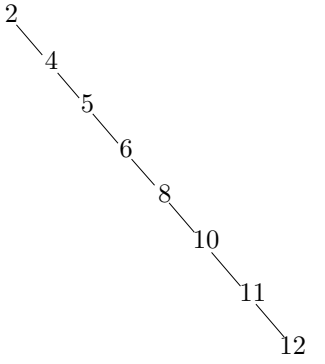
Y la implementación en Pascal es:

```
procedure Insertar(datoNuevo: tElem; var arbol: tArbol);
  {Efecto: se añade ordenadamente a arbol un nodo de contenido
  datoNuevo}
begin
  if arbol = nil then begin
    New(arbol);
    with arbol^ do begin
      hIzdo:= nil;
      hDcho:= nil;
      contenido:= datoNuevo
    end  {with}
  end {then}
  else with arbol^ do
    if datoNuevo < contenido then
      Insertar(datoNuevo,hIzdo)
    else if datoNuevo > contenido then
      Insertar(datoNuevo,hDcho)
    else {No se hace nada: entrada duplicada}
end; {Insertar}
```

La forma resultante de un árbol binario de búsqueda depende bastante del orden en el que se vayan insertando los datos: si éstos ya están ordenados el árbol degenera en una lista (véase la figura 17.11).



Datos: 6, 4, 2, 10, 5, 12, 8, 11



Datos: 2, 4, 5, 6, 8, 10, 11, 12

Figura 17.11.

Supresión de un nodo

Eliminar un nodo en un árbol de búsqueda binaria es la única operación que no es fácil de implementar: hay que considerar los distintos casos que se pueden dar, según el nodo por eliminar sea

1. Una hoja del árbol,
2. Un nodo con un sólo hijo, o
3. Un nodo con dos hijos.

Los dos primeros casos no presentan mayor problema; sin embargo, el tercero requiere un análisis detallado para suprimir el nodo de modo que el árbol resultante siga siendo un árbol binario de búsqueda. En primer lugar hay que situarse en el nodo padre del nodo por eliminar; después procederemos por pasos, analizando qué se necesita hacer en cada caso:

1. Si el nodo por eliminar es una hoja, entonces basta con destruir su variable asociada (usando `Dispose`) y, posteriormente, asignar **nil** a ese puntero.
2. Si el nodo por eliminar sólo tiene un subárbol, se usa la misma idea que al eliminar un nodo interior de una lista: hay que "saltarlo" conectando directamente el nodo anterior con el nodo posterior y desechando el nodo por eliminar.
3. Por último, si el nodo por eliminar tiene dos hijos no se puede aplicar la técnica anterior, simplemente porque entonces habría dos nodos que

conectar y no obtendríamos un árbol binario. La tarea consiste en eliminar el nodo deseado y recomponer las conexiones de modo que se siga teniendo un árbol de búsqueda.

En primer lugar, hay que considerar que el nodo que se coloque en el lugar del nodo eliminado tiene que ser mayor que todos los elementos de su subárbol izquierdo, luego la primera tarea consistirá en buscar tal nodo; de éste se dice que es el *predecesor* del nodo por eliminar (¿en qué posición se encuentra el nodo predecesor?).

Una vez hallado el predecesor el resto es bien fácil, sólo hay que copiar su valor en el nodo por eliminar y desechar el nodo predecesor.

A continuación se presenta un esbozo en seudocódigo del algoritmo en cuestión; la implementación en Pascal del procedimiento se deja como ejercicio indicado.

```
Determinar el número de hijos del nodo N a eliminar
si N no tiene hijos entonces
   eliminarlo
en otro caso si N sólo tiene un hijo H entonces
   Conectar H con el padre de N
en otro caso si N tiene dos hijos entonces
   Buscar el predecesor de N
   Copiar su valor en el nodo a eliminar
   Desechar el nodo predecesor
```

17.4.3 Aplicaciones

Recorrido en anchura de un árbol binario

El uso de colas resulta útil para describir el recorrido en anchura de un árbol. Hemos visto tres formas distintas de recorrer un árbol binario: recorrido en preorden, en inorden y en postorden. El *recorrido primero en anchura* del árbol de la figura 17.10 nos da la siguiente ordenación de nodos: ABCDEFGH. La idea consiste en leer los nodos nivel a nivel, primero la raíz, luego (de izquierda a derecha) los nodos de profundidad 1, los de profundidad 2, etc.

Analicemos cómo trabaja el algoritmo que hay que codificar, para ello seguimos el recorrido en anchura sobre el árbol de la figura 17.10:

1. En primer lugar se lee la raíz de árbol: A.
2. Luego hay que leer los hijos de la raíz, B y C.
3. Después, se leen D y E (hijos de B) y F y G (hijos de C).
4. Finalmente se lee G, el único nodo de profundidad 3.

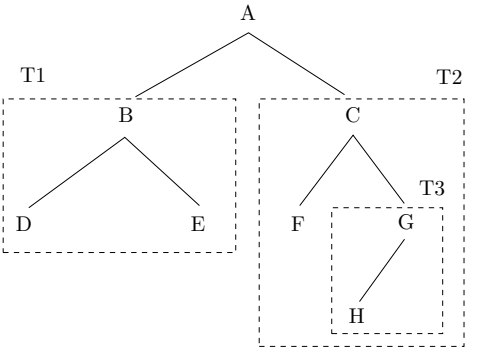


Figura 17.12.

Teniendo en cuenta que los hijos de un nodo son las raíces de sus subárboles hijos se observa que existe una tarea repetitiva, la visita del nodo raíz de un árbol. Lo único que resta es ordenar adecuadamente los subárboles por visitar.

En este punto es donde las colas juegan su papel, pues se va a considerar una cola de subárboles pendientes de visitar, este proceso se representa en la figura 17.12:

1. Tras leer el nodo raíz A se colocan sus dos subárboles hijos en la cola de espera.

2. Se toma el primer árbol de la cola y se visita su raíz, en este caso B, tras lo que se añaden sus subárboles hijos a la cola.

3. ...

La tabla 17.1 muestra, paso a paso, cómo va avanzando el recorrido en anchura y cómo va cambiando el estado de la cola de espera (usando la notación de la figura 17.12 para nombrar a los distintos subárboles con los que se trabaja).

Para la codificación de este recorrido será necesario definir el tipo cola de árboles (el tipo `tCola` por razones de eficiencia) y declarar una variable `enEspera` para almacenar los árboles en espera de ser visitados.

Una primera aproximación en seudocódigo es la siguiente:

Crear una cola con el `arbolDato` *en ella*
Procesar los árboles de esa cola

La primera acción se refina directamente así:

Recorrido	Cola de espera
[ ]	[T]
[A]	[T1, T2]
[A,B]	[T2,D,E]
[A,B,C]	[D,E,F,T3]
[A,B,C,D]	[E,F,T3]
[A,B,C,D,E]	[F,T3]
[A,B,C,D,E,F]	[T3]
[A,B,C,D,E,F,G]	[H]
[A,B,C,D,E,F,G,H]	[ ]

Tabla 17.1.

```
var
  arbolesEnEspera:  cola de árboles
  ...
  CrearCola(arbolesEnEspera);
  PonerEnCola(arbolDato, arbolesEnEspera);
```

donde una cola de árboles se concreta mediante el tipo `tApNodo` (véase el apartado 17.3), siendo sus elementos `tElem` del tipo `tArbol`.

La segunda acción consiste en la serie de pasos de la figura 17.13.

Naturalmente, todo el peso del procedimiento recae en el procesado de los árboles en espera. Para ello hay que leer la raíz del primer árbol de la cola (mientras ésta no esté vacía), sacarlo de ella y añadir al final de la cola sus subárboles hijos. Por tanto, la acción *Procesar los árboles en espera* se puede refinar de la siguiente forma:

mientras `arbolesEnEspera <> nil` *hacer*
  Sacar el primer árbol de la cola
  Procesar su nodo raíz
  Poner en cola los subárboles no vacíos

El refinamiento de cada una de las tareas anteriores es directa haciendo uso de las operaciones descritas para el tipo cola.

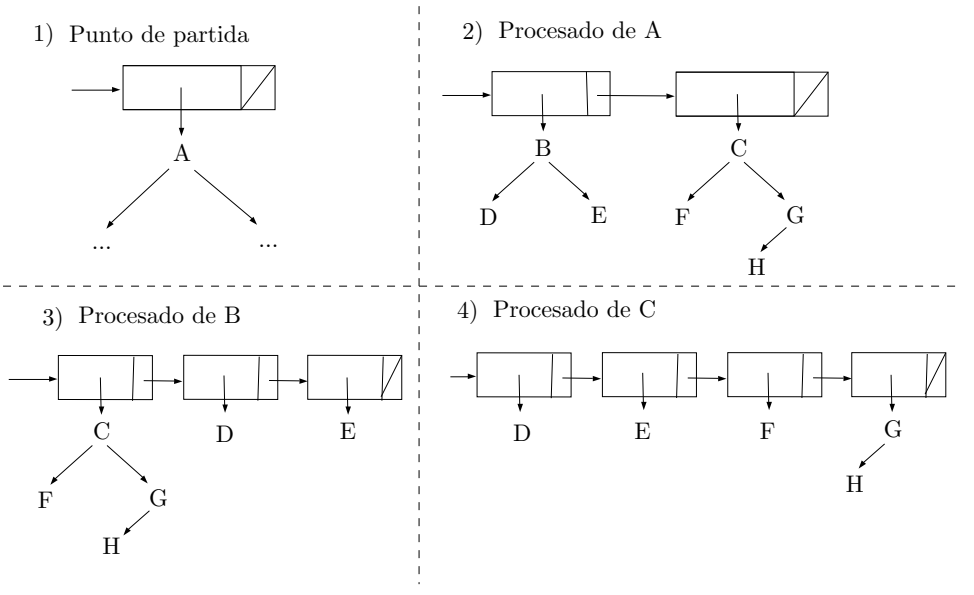


Figura 17.13.

Árboles de expresiones aritméticas

Mediante un árbol binario pueden expresarse las expresiones aritméticas habituales: suma, resta, producto y cociente. En tal representación cada nodo interior del árbol está etiquetado con el símbolo de una operación aritmética (cuyos argumentos son los valores de las expresiones aritméticas que representan sus subárboles hijos), y sólo las hojas están etiquetadas con valores numéricos.

Es interesante observar la relación entre los distintos tipos de recorrido de un árbol binario con las distintas notaciones para las expresiones aritméticas. Por ejemplo, la siguiente expresión dada en notación habitual (esto es, en notación infija):

$$\Big(\big(3 * (4 + 5)\big) - (7 : 2)\Big) : 6$$

tiene el árbol sintáctico de la figura 17.14.

Si se recorre el árbol en *in*orden entonces se obtiene la notación *in*fija (habitual) de la expresión. Si se realiza un recorrido en *post*orden entonces la ordenación de los nodos es la siguiente:

$$3\ 4\ 5\ +\ *\ 7\ 2\ :\ -\ 6\ :$$

que coincide con la expresión en notación *post*fija.

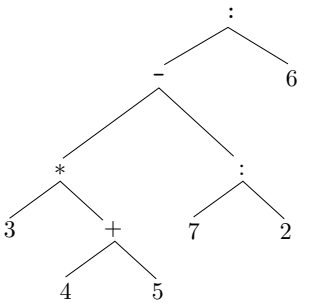


Figura 17.14. Árbol sintáctico de una expresión aritmética.

El lector atento habrá observado que esta notación postfija no coincide con la definida en el apartado 17.2.3; los argumentos de las operaciones aparecen cambiados de orden, esto no es problema con operaciones conmutativas (suma y producto) pero sí en las que no lo son.

Este problema se subsana definiendo un recorrido en postorden en el que se visita antes el hijo derecho que el izquierdo.

Finalmente, si se recorre el árbol en *pre*orden se obtiene una expresión *pre*fija de la expresión, en este caso:

$$: - * 3 + 4\ 5 : 7\ 2\ 6$$

17.5 Otras estructuras dinámicas de datos

En este último apartado damos una idea de otras aplicaciones de la memoria dinámica sin pasar al estudio de la implementación de las mismas.

Listas doblemente enlazadas

La implementación de listas dinámicas mostrada en este capítulo adolece de la imposibilidad de acceder directamente al predecesor de un nodo. Este problema causó la inclusión de varios casos especiales en la implementación de algunas operaciones sobre listas, como la inserción o eliminación de nodos intermedios (véase el apartado 17.1.5).

Una lista *doblemente enlazada* o *de doble enlace* se define de modo similar al de las listas de enlace simple, sólo que cada nodo dispone de dos punteros que apuntan al nodo anterior y al nodo siguiente:

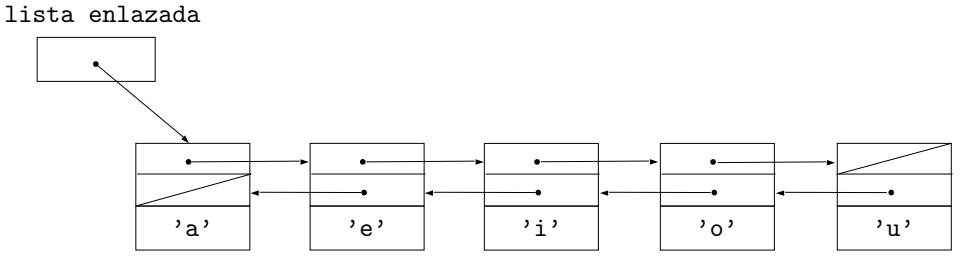


Figura 17.15. Una lista doblemente enlazada.

```
type
  tElem = char;  {o lo que corresponda}
  tListaDobleEnlace = ^tNodo;
  tNodo = record
    contenido : tElem;
    anterior, siguiente: tListaDobleEnlace
  end; {tNodo}
```

La figura 17.15 representa una lista doblemente enlazada. Con los nodos primero y último de la lista se pueden tomar dos posturas:

1. Considerar que el anterior del primer nodo es **nil**, y que el siguiente al último también es **nil**.

   En este caso resulta útil considerar la lista como un registro con dos componentes: principio y final. Del mismo modo que en la implementación de `tCola`.

2. Considerar que el anterior del primero es el último y que el siguiente al último es el primero.

   En este caso, estrictamente hablando, no se obtiene una lista, sino un *anillo* o *lista circular* que carece de principio y de final.

Un análisis más profundo de esta estructura rebasa los límites de este libro; no obstante, en el apartado de comentarios bibliográficos se citan algunas referencias con las que profundizar en el estudio de este tipo de datos.

Árboles generales

En el apartado 17.4 se han estudiado los árboles binarios; esta estructura se puede generalizar a estructuras en las que los nodos pueden tener más de dos subárboles hijos. Esta generalización puede hacerse de dos formas distintas:

1. Pasando a *árboles n-arios*, en los que cada nodo tiene a lo sumo n subárboles hijos.

   Un árbol *árbol n-ario* se puede representar como un árbol de registros de n componentes, o bien como un vector de subárboles.

2. Considerando *árboles generales*, en los que no existe limitación en el número de subárboles hijo que puede tener.

   Los árboles generales se implementan mediante un árbol de listas (puesto que no se sabe el número máximo de hijos de cada nodo).

La elección de un tipo de árbol o de otro depende del problema particular que se esté tratando: el uso de registros acelera el acceso a un hijo arbitrario; sin embargo, un número elevado de registros puede consumir buena parte de la memoria disponible. El otro enfoque, el de un árbol general, resulta apropiado para evitar el derroche de memoria si no se usa la mayoría de los campos de los registros con la contrapartida de un mayor tiempo de acceso a los nodos.

Entre las aplicaciones típicas de los árboles generales se encuentran los árboles de juegos o los árboles de decisión. Veamos brevemente qué se entiende por un árbol de juego (de mesa, como, por ejemplo, las damas o el ajedrez):

La raíz de un árbol de juegos es una posición de las fichas en el tablero; el conjunto de sus hijos lo forman las distintas posiciones accesibles en un movimiento desde la posición anterior. Si no es posible realizar ningún movimiento desde una posición, entonces ese nodo no tiene descendientes, es decir, es una hoja; como ejemplo de árbol de juegos, en la figura 17.16 se muestra un árbol para el juego del tres en raya, representando una estrategia que permite ganar siempre al primero en jugar.

Conviene saber que no es corriente razonar sobre juegos desarrollando explícitamente todo el árbol de posibilidades, sino que suele "recorrerse" implícitamente (hasta cierto punto) al efectuarse llamadas de subprogramas recursivos.

17.6 Ejercicios

1. Escriba una versión iterativa de la función `longitud` de una lista.
2. Se denomina *palíndromo* una palabra o frase que se lee igual de izquierda a derecha que de derecha a izquierda. Por ejemplo,

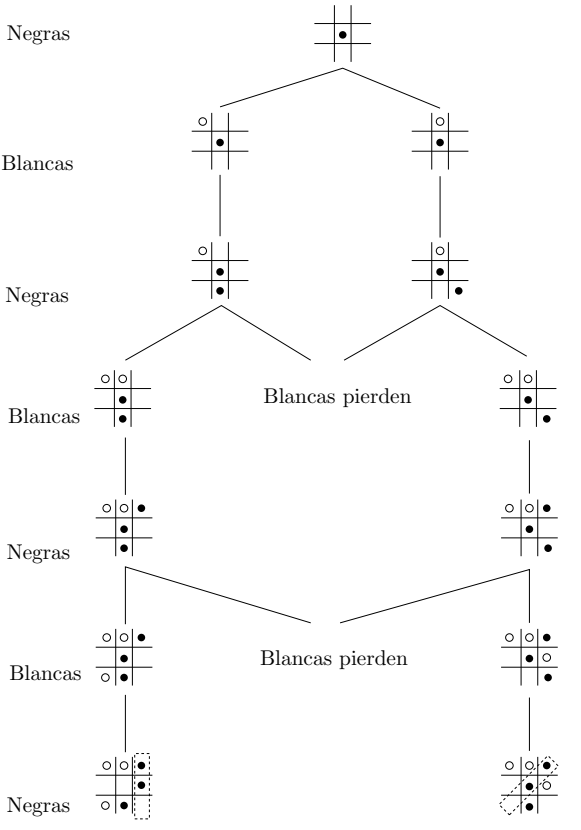


Figura 17.16. Estrategia ganadora para el juego del tres en raya.

DABALE ARROZ A LA ZORRA EL ABAD

Escriba un programa que lea una cadena de caracteres (terminada por el carácter de fin de línea) y determine si es o no un palíndromo. (Indicación: un método sencillo consiste en crear una lista de letras y su inversa y compararlas para ver si son iguales).

3. Completar el programa de simulación de colas, calculando el tiempo que permanece la caja desocupada y el tiempo medio de espera de los clientes.
4. La función de búsqueda de un dato en un árbol binario de búsqueda se presentó de forma recursiva. Escribir una versión iterativa de dicha función.
5. Escriba un procedimiento que halle el nodo predecesor de un nodo de un árbol binario de búsqueda. Úsese para escribir una codificación completa de la eliminación de un nodo en un árbol de búsqueda binaria.
6. Completar la codificación en Pascal del recorrido en anchura de un árbol. Para el procesado de los nodos visitados limítese a imprimir el contenido de los nodos.
7. Implemente una versión iterativa del procecimiento `PonerEnCola`.
8. Escriba la versión iterativa de un procedimiento recursivo genérico.
9. Escriba una función que evalúe expresiones aritméticas escritas en notación postfija.

17.7 Referencias bibliográficas

En [DL89] se hace un estudio detallado de de las principales estructuras dinámicas de datos con numerosos ejemplos y aplicaciones, siguiendo el mismo estilo que en la primera parte de este texto [DW89].

Una referencia obligada es el clásico libro de N. Wirth [Wir86], que dedica su capítulo 4 al estudio de las estructuras dinámicas de datos. Dentro de ellas hay que destacar el tratamiento de los distintos tipos de árboles: equilibrados, de búsqueda, generales y otros, con interesantes aplicaciones.

Esta parte de nuestro libro debe entenderse como una introducción, ya que las estructuras de datos que pueden construirse con punteros son variadísimas (árboles, grafos, tablas, conjuntos, matrices de gran tamaño y bases de datos, por citar algunos) y pueden llegar a ser muy complejos. Su estudio en detalle corresponde a cursos posteriores. Por citar algunos textos en español sobre el tema destacamos [AM88] y [AHU88].

Finalmente, es obligado mencionar que la idea de los punteros con referencias indirectas de los datos es ampliamente explotada por el lenguaje C y sus extensiones. Precisamente ese uso extensivo es una razón de peso para lograr programas eficientes. Entre las muchas referencias sobre este lenguaje, citamos [KR86].

Tema VI

Aspectos avanzados de programación

Capítulo 18

Complejidad algorítmica

Con frecuencia, un problema se puede resolver con varios algoritmos, como ocurre, por ejemplo, en los problemas de ordenación y búsqueda de vectores (véase el capítulo 15).

En este capítulo se estudian criterios (presentados escuetamente en el apartado 1.3.3) que permiten al programador decidir cuál de los posibles algoritmos que resuelven un problema es más eficiente. En general, la eficiencia se puede entender como una medida de los recursos empleados por un algoritmo en su ejecución. En particular, usualmente se estudia la eficiencia de un algoritmo en tiempo (de ejecución), espacio (de memoria) o número de procesadores (en algoritmos implementados en arquitecturas paralelas). Como se vio en el apartado 1.3.3, el estudio de la complejidad algorítmica trata de resolver este importantísimo aspecto de la resolución de problemas.

Los criterios utilizados por la complejidad algorítmica no proporcionan medidas absolutas, como podría ser el tiempo total en segundos empleado en la ejecución del programa que implementa el algoritmo, sino medidas relativas al

tamaño del problema. Además, estas medidas son independientes del computador sobre el que se ejecute el algoritmo.

Nuestro objetivo en este tema es proporcionar las herramientas necesarias para el cálculo de la complejidad de los algoritmos. De esta forma, en caso de disponer de más de un algoritmo para solucionar un problema, tendremos elementos de juicio para decidir cuál es mejor desde el punto de vista de la eficiencia.

Con frecuencia, no es posible mejorar simultáneamente la eficiencia en tiempo y en memoria, buscándose entonces un algoritmo que tenga una complejidad razonable en ambos aspectos. Actualmente, y gracias a los avances de la técnica, quizás es más importante el estudio de la complejidad en el tiempo, ya que la memoria de un computador puede ser ampliada fácilmente, mientras que el problema de la lentitud de un algoritmo suele ser más difícil de resolver.

Además, el cálculo de las complejidades en tiempo y en espacio se lleva a cabo de forma muy similar. Por estas razones, se ha decidido dedicar este capítulo esencialmente al estudio de la complejidad en el tiempo.

18.1 Conceptos básicos

El tiempo empleado por un algoritmo se mide en "pasos"

Para medir el tiempo empleado por un algoritmo se necesita una medida adecuada. Claramente, si se aceptan las medidas de tiempo físico, obtendremos unas medidas que dependerán fuertemente del computador utilizado: si se ejecuta un mismo algoritmo con el mismo conjunto de datos de entrada en dos computadores distintos (por ejemplo en un PC-XT y en un PC-Pentium) el tiempo empleado en cada caso difiere notablemente. Esta solución llevaría a desarrollar una teoría para cada computador, lo que resulta poco práctico.

Por otro lado, se podría contar el número de instrucciones ejecutadas por el algoritmo, pero esta medida dependería de aspectos tales como la habilidad del programador o, lo que es más importante, del lenguaje de programación en el que se implemente, y tampoco se debe desarrollar una teoría para cada lenguaje de programación.

Se debe buscar entonces una medida abstracta del tiempo que sea independiente de la máquina con que se trabaje, del lenguaje de programación, del compilador o de cualquier otro elemento de *hardware* o *software* que influya en el análisis de la complejidad en tiempo. Una de las posibles medidas, que es la empleada en este libro y en gran parte de la literatura sobre complejidad, consiste en contar *el número de pasos* (por ejemplo, operaciones aritméticas, comparaciones y asignaciones) que se efectúan al ejecutarse un algoritmo.

El coste depende de los datos

Considérese el problema de decidir si un número natural es par o impar. Es posible usar la función `Odd` predefinida en Pascal, que permite resolver el problema en tiempo constante.

Una segunda opción es emplear el algoritmo consistente en ir restando 2 repetidamente mientras que el resultado de la sustracción sea mayor que 1, y finalmente comprobar el valor del resto. Es fácil comprobar que se realizan `n div 2` restas, lo que nos indica que, en este caso, el tiempo empleado depende del dato original `n`. Normalmente, el tiempo requerido por un algoritmo es función de los datos, por lo que se expresa como tal: así, escribimos $T(n)$ para representar la complejidad en tiempo para un dato de tamaño n.

Este tamaño de entrada `n` depende fuertemente del tipo de problema que se va a estudiar. Así, por ejemplo, en el proceso de invertir el orden de los dígitos de un número natural

$$(4351, 0) \rightsquigarrow (435, 1) \rightsquigarrow (43, 15) \rightsquigarrow (4, 153) \rightsquigarrow (0, 1534)$$

no importa el número en cuestión: el dato relevante para ver cuánto tiempo se tarda es la longitud del número que se invierte. Otro ejemplo: para sumar las componentes de un vector de números reales, el tiempo empleado depende del número `n` de componentes del vector. En este caso, se puede decir ambas cosas sobre el coste:

- Que el coste de invertir un número, dígito a dígito, es lineal con respecto a su longitud (número de cifras)
- Que el coste es la parte entera de $\log_{10}(n)+1$, o sea, una función logarítmica, siendo n el dato.

Otra situación ejemplar se da en el caso del algoritmo de *suma lenta* (véase el apartado 1.2.1) de dos números naturales `a` y `b`:

```
while b > 0 do begin
  a:= a + 1;
  b:= b - 1
end {while}
```

Como se puede observar, el coste del algoritmo depende únicamente del segundo parámetro, ya que se ejecutan exactamente `b` iteraciones del cuerpo del bucle. Es, por tanto, un algoritmo de complejidad lineal con respecto a `b`, es decir, $T(\mathtt{a}, \mathtt{b}) = \mathtt{b}$. Conclusión: no siempre todos los datos son importantes de cara a la complejidad.

En definitiva, a la hora de elegir el tamaño de los datos de entrada `n` de un algoritmo, conviene que represente la parte o la característica de los datos que influye en el coste del algoritmo.

El coste esperado, el mejor y el peor

Otro aspecto interesante de la complejidad en tiempo puede ilustrarse analizando el algoritmo de búsqueda secuencial ordenada estudiado en el apartado 15.1.2, en el que se recorre un vector (ordenado crecientemente) desde su primer elemento, hasta encontrar el elemento buscado, o hasta que nos encontremos un elemento en el vector que es mayor que el elemento `elem` buscado. La implementación en Pascal de dicho algoritmo (ya mostrada en el citado apartado) es la siguiente:

```
const
  N = 100; {tamaño del vector}
type
  tIntervalo = 0..N;
  tVector = array[1..N] of integer;

function BusquedaSecOrd(v: tVector; elem: integer): tIntervalo;
  {PreC.: v está ordenado crecientemente, sin repeticiones}
  {Dev. 0 (si elem no está en v) ó i (si v[i] = elem)}
  var
    i: tIntervalo;
begin
  i:= 0;
  repeat
      {Inv.: v[j] ≠ elem ∀j,1 ≤ j ≤ i}
    i:= i + 1
  until (v[i] >= elem) or (i = N);
    {v[i] = elem o v[j] ≠ elem ∀j,1 ≤ j ≤ N}
  if v[i] = elem then {se ha encontrado el valor elem}
    BusquedaSecOrd:= i
  else
    BusquedaSecOrd:= 0
end; {BusquedaSecOrd}
```

Intuitivamente se puede ver que, si se tiene la buena fortuna de encontrar el elemento al primer intento, el tiempo es, digamos, de un paso (un intento). En el peor caso (cuando el elemento `elem` buscado es mayor o igual que todos los elementos del vector), se tendrá que recorrer todo el vector `v`, invirtiendo `n` pasos. Informalmente, se podría pensar que en un caso "normal", se recorrería la "mitad" del vector (`n/2` pasos).

Como conclusión se puede afirmar que en algunos algoritmos, la complejidad no depende únicamente del tamaño del parámetro, sino que intervienen otros factores que hacen que la complejidad varíe de un caso a otro. Para distinguir esas situaciones se habla de coste *en el mejor caso*, *en el peor caso* y *en el caso medio*:

- $T_{\text{máx}}(\texttt{n})$, expresa la complejidad en *el peor caso*, esto es, el tiempo máximo que un algoritmo puede necesitar para una entrada de tamaño `n`.

- $T_{\text{mín}}(\texttt{n})$, expresa la complejidad en *el mejor caso*, esto es, el tiempo mínimo que un algoritmo necesita para una entrada de tamaño `n`.

- $T_{\text{med}}(\texttt{n})$, expresa la complejidad en *el caso medio*, esto es, el tiempo medio que un algoritmo necesita para una entrada de tamaño `n`. Generalmente, se suele suponer que todas las secuencias de entradas son equiprobables. Por ejemplo, en los algoritmos de búsqueda, se considerará que `elem` puede estar en cualquier posición del vector con idéntica probabilidad, es decir, con probabilidad $\frac{1}{n}$.

Generalmente, la complejidad en el mejor caso es poco representativa y la complejidad en el caso medio es difícil de calcular por lo que, en la práctica, se suele trabajar con el tiempo para el peor caso por ser una medida significativa y de cálculo factible en general. No obstante, como ejemplo, se calculan a continuación las tres medidas para el algoritmo de búsqueda secuencial ordenada.

Para fijar ideas, vamos a hallar el coste en términos de los tiempos empleados para las operaciones de sumar (s), realizar una comparación (c) y realizar una asignación (a) por un computador cualquiera. El coste en pasos es más sencillo, ya que basta con dar el valor unidad a cada una de esas operaciones.

Las tres medidas de coste mencionadas se calculan como sigue:

- $T_{\text{mín}}$: Este tiempo mínimo se alcanzará cuando `v[1]` $\geq$ `elem`. En tal caso se necesita una asignación para iniciar la variable `i`, una suma y una asignación para incrementar el valor de `i`, dos comparaciones en el bucle **repeat**, otro test más para `v[i] = elem` y, finalmente, una asignación a la función `BúsquedaSecOrd`. Por lo tanto:

$$T_{\text{mín}}(\texttt{n}) = 3a + 3t + s$$

que es constante, lo que abreviamos así:

$$T_{\text{mín}}(\texttt{n}) = k$$

- $T_{\text{máx}}$: Este tiempo máximo se alcanzará cuando `v[n]` $\leq$ `elem`. Por lo tanto:

$$T_{\text{máx}}(\texttt{n}) \;=\; a + n(s + 2t + a) + t + a \;=\; k_1 n + k_2$$

- T_{med}: Supóngase que es igualmente probable necesitar 1 vuelta, 2 vueltas,..., `n` vueltas para encontrar el elemento buscado.[1] Además, recordemos que el tiempo empleado por el algoritmo cuando para en la posición j-ésima del vector es

$$T(j) = k_1 j + k_2$$

según lo dicho en el apartado anterior. Entonces, se tiene que:[2]

$$\begin{aligned} T_{\text{med}}(n) &= \sum_{j=1}^{n} T_j P(\textit{parar en la posición } j-\textit{ésima}) \\ &= \sum_{j=1}^{n} (k_1 j + k_2)\frac{1}{n} \\ &= \frac{k_1}{n}\frac{n+1}{2} + k_2 \\ &= c_1 n + c_2 \end{aligned}$$

de forma que también es lineal con respecto a `n`.

Por supuesto, $T_{\text{med}}(\texttt{n}) < T_{\text{máx}}(\texttt{n})$, como se puede comprobar comparando los valores de k_1 y c_1, lo que se deja como ejercicio al lector.

También importa el gasto de memoria

Por otra parte, el estudio de la implementación de la función `Sumatorio` nos sirve para ver cómo el coste en memoria también debe tenerse en cuenta al analizar algoritmos.

Esta función, que calcula la suma de los `n` primeros números naturales, siendo `n` el argumento de entrada, puede ser implementada de forma natural con un algoritmo recursivo, resultando el siguiente código en Pascal:

```
function Sumatorio(n: integer): integer;
  {PreC.: n ≥ 0}
  {Dev. ∑_{i=0}^{n} i}
```

[1] Se tiene esta situación, por ejemplo, cuando la secuencia ordenada de los `n` elementos del vector se ha escogido equiprobablemente entre el dominio `integer`, así como el elemento `elem` que se busca.

[2] Notaremos con P a la probabilidad.

```
begin
  if n = 0 then
    Sumatorio:= 0
  else
    Sumatorio:= n + Sumatorio(n-1)
end;  {Sumatorio}
```

Al calcular el espacio de memoria que ocupa una llamada a esta función ha de tenerse en cuenta que, por su naturaleza recursiva, se generan nuevas llamadas a `Sumatorio` (exactamente `n` llamadas). En cada llamada se genera una tabla de activación (véase el apartado 10.2) del tamaño de un entero (el parámetro `n`), es decir, de tamaño constante. En consecuencia, podemos afirmar que la función `Sumatorio` tiene un coste proporcional a `n`.

Pero la suma de los `n` primeros enteros puede calcularse también de forma intuitiva empleando un algoritmo iterativo. Su sencilla implementación es la siguiente:

```
function SumatorioIter(n:  integer):  integer;
  {PreC.:  n ≥ 0}
  {Dev.  ∑_{i=0}^{n} i}
  var
    suma, i:  integer;
begin
  suma:= 0;
  for i:= 0 to n do
    suma:= suma + i;
  SumatorioIter:= suma
end;  {SumatorioIter}
```

En este caso, una llamada a `SumatorioIter`, al no generar otras llamadas sucesivas, consume un espacio de memoria constante: exactamente el necesario para su parámetro y para las dos variables locales, todos de tipo `integer`. Piense el lector en la diferencia de espacio requerida por ambas funciones para $\texttt{n} = 1000$, por ejemplo.

Con estos ejemplos podemos concluir que, a la hora del análisis de algoritmos, es fundamental realizar un estudio de la eficiencia, destacando como aspecto más importante la complejidad en tiempo, y en segundo lugar, la complejidad en espacio.

Lo importante es el comportamiento asintótico

Es un hecho evidente que datos de un tamaño reducido van a tener asociados, en general, tiempos cortos de ejecución. Por eso, es necesario estudiar el com-

portamiento de éstos con datos de un tamaño considerable, que es cuando los costes de los distintos algoritmos pueden tener una diferenciación significativa.

Para entender mejor la importancia del orden de complejidad, resulta muy ilustrativo observar cómo aumenta el tiempo de ejecución de algoritmos con distintos órdenes. En todos ellos, n representa el tamaño de los datos y los tiempos están expresados en segundos, considerando un computador que realiza un millón de operaciones por segundo:

$T(n)$ / n	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	$3.32 \cdot 10^{-6}$	10^{-5}	$3.32 \cdot 10^{-5}$	10^{-4}	0.001	0.001024	3.6288
50	$5.64 \cdot 10^{-6}$	$5 \cdot 10^{-5}$	$2.82 \cdot 10^{-4}$	0.0025	0.125	intratable	intratable
100	$6.64 \cdot 10^{-6}$	10^{-4}	$6.64 \cdot 10^{-4}$	0.01	1	intratable	intratable
10^3	10^{-5}	0.001	0.01	1	1000	intratable	intratable
10^4	$1.33 \cdot 10^{-5}$	0.01	0.133	100	10^6	intratable	intratable
10^5	$1.66 \cdot 10^{-5}$	0.1	1.66	10^4	intratable	intratable	intratable
10^6	$2 \cdot 10^{-5}$	1	19.93	10^6	intratable	intratable	intratable

18.2 Medidas del comportamiento asintótico

18.2.1 Comportamiento asintótico

Como se ha visto en el apartado anterior, la complejidad en tiempo de un algoritmo es una función $T(n)$ del tamaño de entrada del algoritmo. Pues bien, es el orden de dicha función (constante, logarítmica, lineal, exponencial, etc.) el que expresa el comportamiento dominante para datos de gran tamaño, como se ilustra en el ejemplo que se presenta a continuación.

Supóngase que se dispone de cuatro algoritmos para solucionar un determinado problema, cuyas complejidades son respectivamente, lineal ($8n$), cuadrática ($2\ n^2$), logarítmica ($20 \log_2 n$) y exponencial (e^n). En la figura 18.1 se puede observar cómo sus tiempos relativos de ejecución no son excesivamente diferentes para datos de un tamaño pequeño (entre 1 y 5).

Sin embargo, las gráficas de la figura 18.2 confirman que es realmente el orden de la función de complejidad el que determina el comportamiento para tamaños de entrada grandes, reteniendo únicamente la parte relevante de una función (de coste) para datos de gran tamaño.

A la vista de esto, es evidente que el aspecto importante de la complejidad de algoritmos es su comportamiento asintótico, ignorando otros detalles menores por ser irrelevantes.

Para ello, es preciso formalizar el estudio del orden de complejidad mediante medidas que ayuden a determinar el comportamiento asintótico del coste.

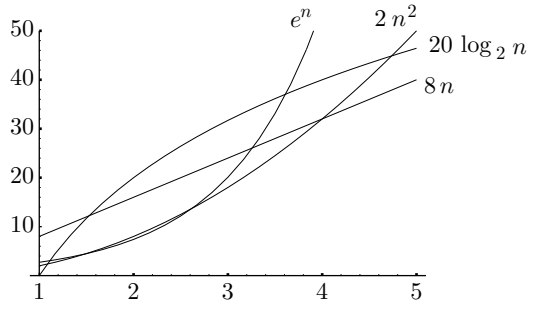


Figura 18.1.

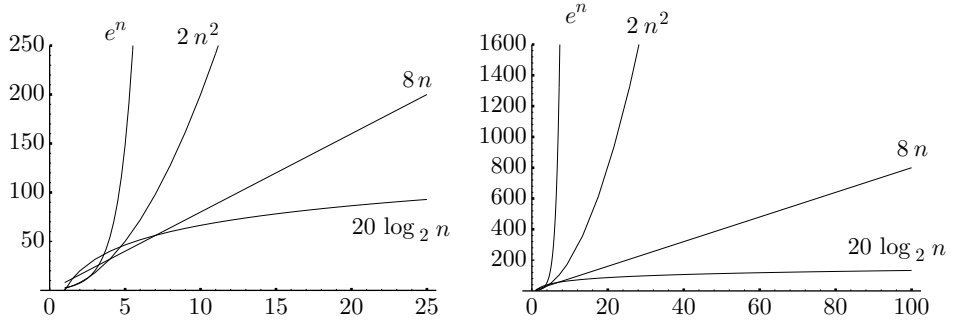


Figura 18.2.

Entre estas medidas destaca la notación O mayúscula,[3] la notación Ω y la notación Θ.

18.2.2 Notación O mayúscula (una cota superior)

Definición: Sean $f, g : \mathbb{Z}^+ \longrightarrow \mathbb{R}^+$. Se dice que $f \in O(g)$ o que f *es del orden de* g si existen constantes $n_0 \in \mathbb{Z}^+$ y $\lambda \in \mathbb{R}^+$ tales que

$$f(n) \leq \lambda g(n) \qquad \text{para todo } n \geq n_0$$

Con la notación[4] $f \in O(g)$ se expresa que la función f no crece más deprisa que alguna función proporcional a g. Esto es, se acota superiormente el comportamiento asintótico de una función salvo constantes de proporcionalidad. Veamos algunos ejemplos:

- Para el algoritmo de búsqueda secuencial ordenada,

  $$T_{\text{máx}}(n) = k_1 n + k_2 \in O(n)$$

  (lo que se ve tomando cualquier $\lambda > k_1$ y $n_0 > \frac{k_2}{\lambda - k_1}$).
  Para este mismo algoritmo se tiene además que $T_{\text{med}}(n) = c_1 n + c_2 \in O(n)$, y para el sumatorio recursivo se cumple que $S(n) = n + 1 \in O(n)$.

- Todas las funciones de tiempo constante son $O(1)$:

  $$f(n) = k \in O(1)$$

  lo que se ve tomando $\lambda = k$ y cualquier $n_0 \in \mathbb{Z}^+$. En este caso están $T_{\text{mín}}(n) = k$ para la búsqueda secuencial ordenada, y $S(n) = 3$ para el sumatorio iterativo.

- $15n^2 \in O(n^2)$

Como esta notación expresa una cota superior, siempre es posible apuntar alto a la hora de establecerla. Por ejemplo, se puede decir que los algoritmos estudiados hasta ahora son $O(n!)$. Naturalmente, esta imprecisión es perfectamente inútil, por lo que se debe procurar que la función g sea lo más "próxima" posible a f; es decir, se debe buscar una cota superior lo menor posible. Así, por ejemplo, aunque $(5n + 3) \in O(n^2)$, es más preciso decir que $(5n + 3) \in O(n)$.

Para formalizar esta necesidad de precisión, se usan otras medidas del comportamiento asintótico.

[3] Debido a que la expresión inglesa que se utiliza para esta notación es *Big-Oh*, también se conoce como notación O grande.

[4] En lo sucesivo, emplearemos las dos notaciones $f \in O(g)$ y $f(n) \in O(g(n))$ indistintamente, según convenga.

18.2.3 Notación Ω mayúscula (una cota inferior)

Definición: Sean $f, g : \mathbb{Z}^+ \longrightarrow \mathbb{R}^+$. Se dice que $f \in \Omega(g)$ si existen constantes $n_0 \in \mathbb{Z}^+$ y $\lambda \in \mathbb{R}^+$ tales que

$$f(n) \geq \lambda g(n) \qquad \text{para todo } n \geq n_0$$

Con la notación $f \in \Omega(g)$ se expresa que la función f crece más deprisa que alguna función proporcional a g. Esto es, se acota inferiormente el comportamiento asintótico de una función, salvo constantes de proporcionalidad. Dicho de otro modo, con la notación $f \in \Omega(g)$ se indica que la función f necesita para su ejecución un tiempo mínimo dado por el orden de la función g.

En los ejemplos anteriores se puede comprobar que:

- Para el algoritmo de búsqueda secuencial ordenada, $T_{\text{máx}}(n) \in \Omega(n)$.
- $3n + 1 \in \Omega(n)$.
- $3n + 1 \in \Omega(1)$.
- $15n^2 \in \Omega(n^2)$.
- $15n^2 \in \Omega(n)$.

Comparando las definiciones anteriores, se tiene que

$$f \in O(g) \Leftrightarrow g \in \Omega(f)$$

18.2.4 Notación Θ mayúscula (orden de una función)

Definición: Sean $f, g : \mathbb{Z}^+ \longrightarrow \mathbb{R}^+$. Se dice que $f \in \Theta(g)$ si $f \in O(g)$ y $g \in O(f)$; esto es, si $f \in O(g) \cap \Omega(g)$. Al conjunto $\Theta(g)$ se le conoce como el *orden exacto de g*.

Con la notación Θ se expresa que la funciones f y g tienen el mismo "grado" de crecimiento, es decir, que $0 < \lim_{x\to\infty} \frac{f(x)}{g(x)} < \infty$. Ejemplos:

- Para el algoritmo de búsqueda secuencial ordenada, $T_{\text{máx}}(n) \in \Theta(n)$.
- $3n + 1 \in \Theta(n)$.
- $15n^2 \in \Theta(n^2)$.

18.2.5 Propiedades de O, Ω y Θ

Entre las propiedades más importantes de la notaciones O mayúscula, Ω y Θ cabe destacar las que se describen a continuación. En ellas se utiliza el símbolo Δ si la propiedad es cierta para las tres notaciones, y se asume que

$$f, g, f_1, f_2 : \mathbb{Z}^+ \longrightarrow \mathbb{R}^+$$

Reflexividad: $f \in \Delta(f)$.

Escalabilidad: Si $f \in \Delta(g)$ entonces $f \in \Delta(k \cdot g)$ para todo $k \in \mathbb{R}^+$.

Una consecuencia de ello es que, si $a, b > 1$ se tiene que $O(\log_a n) = O(\log_b n)$. Por ello, no hace falta indicar la base: $O(\log\ n)$

Transitividad: Si $f \in \Delta(g)$ y $g \in \Delta(h)$ se tiene que $f \in \Delta(h)$.

Simetría: Si $f \in \Theta(g)$ entonces $g \in \Theta(f)$

(Obsérvese que las otras notaciones no son simétricas y trátese de dar un contraejemplo.)

Regla de la suma: Si $f_1 \in O(g_1)$ y $f_2 \in O(g_2)$ entonces $f_1 + f_2 \in O\left(\text{máx}(g_1, g_2)\right)$ siendo $\text{máx}(g_1, g_2)(n) = \text{máx}\left(g_1(n), g_2(n)\right)$.

Junto con la escalabilidad, esta regla se generaliza fácilmente así: si $f_i \in O(f)$ para todo $i = 1, \ldots k$, entonces $c_1 f_1 + \ldots + c_k f_k \in O(f)$.

Otra consecuencia útil es que si $p_k(n)$ es un polinomio de grado k, entonces $p_k(n) \in O(n^k)$.

Regla del producto: Si $f_1 \in \Delta(g_1)$ y $f_2 \in \Delta(g_2)$ entonces $f_1 \cdot f_2 \in O(g_1 \cdot g_2)$.

Una consecuencia de ello es que, si $p < q$, entonces $O(n^p) \subset O(n^q)$.

Regla del sumatorio: Si $f \in O(g)$ y la función g es creciente,

$$\sum_{i=1}^{n} f(i) \in O\left(\int_1^{n+1} g(x)\ dx\right)$$

.

Una consecuencia útil es la siguiente: $\sum_{i=1}^{n} i^k \in O(n^{k+1})$.

Estas propiedades no deben interpretarse como meras fórmulas desprovistas de significado. Muy al contrario, expresan la idea de fondo de las medidas asintóticas, que consiste en ver la parte relevante de una función de coste pensando en datos de gran tamaño. Gracias a ello, podemos simplificar considerablemente las funciones de coste sin peligro de pérdida de información (para datos grandes, se entiende). Por ejemplo:

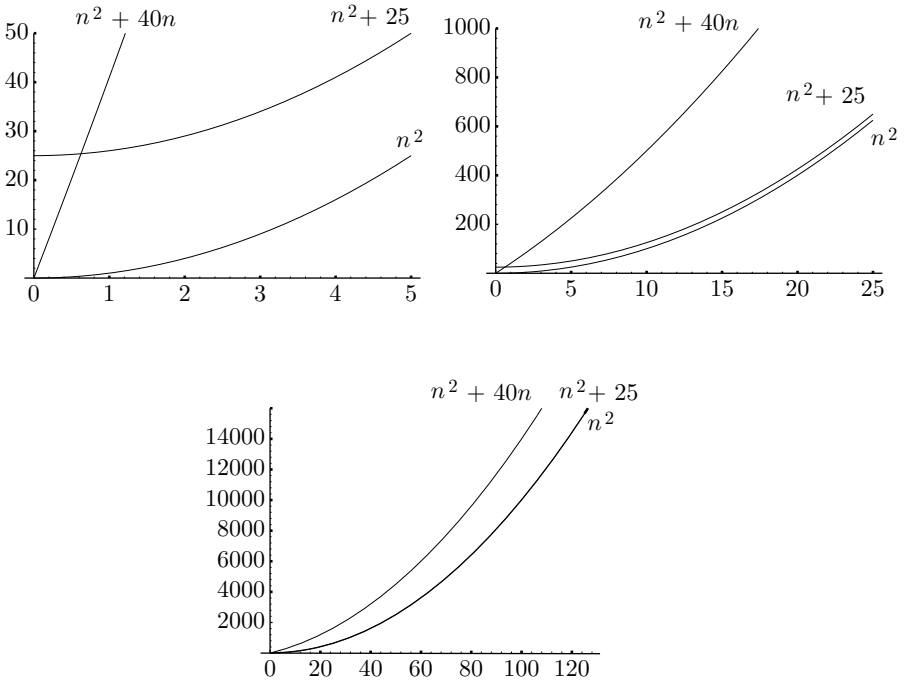


Figura 18.3.

$$c_1 n^2 + c_2 n + c_3 \sim c_1 n^2 \sim n^2$$

Efectivamente, para datos grandes las funciones con el mismo orden de complejidad se comportan esencialmente igual. Además, en la práctica es posible omitir los coeficientes de proporcionalidad: de hecho, no afectan a las medidas estudiadas. Las dos simplificaciones se justifican en las gráficas de la figura 18.3.

18.2.6 Jerarquía de órdenes de frecuente aparición

Existen algoritmos de complejidad *lineal* con respecto a los datos de entrada ($T(n) = c_1 n + c_2$). También existen algoritmos de complejidad *constante* ($T(n) = c$), independientemente de los datos de entrada. El método de intercambio directo para ordenar arrays tiene un coste *cuadrático*. Otras funciones de coste son *polinómicas* de diversos grados, *exponenciales*, *logarítmicas*, etc.

Estos diferentes comportamientos asintóticos se pueden ordenar de menor a mayor crecimiento. Aunque no pretendemos dar una lista exhaustiva, la siguiente cadena de desigualdades puede orientar sobre algunos de los órdenes de coste más usuales:[5]

$$1 \ll \log(n) \ll n \ll n\log(n) \ll n^2 \ll n^3 \ll \cdots \ll 2^n \ll n!$$

A pesar de las apariencias, la relación "ser del orden de" no es una relación de orden total: existen pares de funciones tales que ninguna de las dos es una cota superior de la otra. Por ejemplo, las funciones

$$f(n) = \begin{cases} 1 & \text{si } n \text{ es par} \\ n & \text{si } n \text{ es impar} \end{cases} \quad g(n) = \begin{cases} n & \text{si } n \text{ es par} \\ 1 & \text{si } n \text{ es impar} \end{cases}$$

no verifican $f \in O(g)$ ni $g \in O(f)$.

18.3 Reglas prácticas para hallar el coste de un programa

18.3.1 Tiempo empleado

En este apartado se dan reglas generales para el cálculo de la complejidad en tiempo en el peor caso de los programas escritos en Pascal. Para dicho cálculo, como es natural, se debe tener en cuenta el coste en tiempo de cada una de las diferentes instrucciones de Pascal, y esto es lo que se detalla a continuación.

Instrucciones simples

Se considera que se ejecutan en tiempo constante:

- La evaluación de las expresiones aritméticas (suma, resta, producto y división) siempre que los datos sean de tamaño constante, así como las comparaciones de datos simples.
- Las instrucciones de asignación, lectura y escritura de datos simples.
- Las operaciones de acceso a una componente de un array, a un campo de un registro y a la siguiente posición de un archivo.

Todas esas operaciones e instrucciones son $\Theta(1)$.

[5] La notación $\ll$ representa la relación *de orden menor que*.

Composición de instrucciones

Suponiendo que las instrucciones I_1 e I_2 tienen como complejidades en el peor caso $T_{I_1}(n)$ y $T_{I_2}(n)$, respectivamente, entonces el coste de la composición de instrucciones $(I_1; I_2)$ en el peor caso es

$$T_{I_1;I_2}(n) = T_{I_1}(n) + T_{I_2}(n)$$

que, aplicando la regla de la suma, es el máximo entre los costes $T_{I_1}(n)$ y $T_{I_2}(n)$.

Instrucciones de selección

En la instrucción condicional,

if *condición* **then** I_1 **else** I_2

siempre se evalúa la condición, por lo que su coste debe agregarse al de la instrucción que se ejecute. Puesto que se está estudiando el coste en el peor caso, se tendrá en cuenta la más costosa. Con todo esto, la complejidad de la instrucción **if-then-else** es:

$$T_{condición}(n) + máx(T_{I_1}(n), T_{I_2}(n))$$

Análogamente, la instrucción de selección por casos

case *expresión* **of**
  caso1: I_1;
  ...
  casoL: I_L
end; {case}

requiere evaluar la expresión y una instrucción, en el peor caso la más costosa:

$$T_{expresión}(n) + máx(T_{I_1}(n), \ldots, T_{I_L}(n))$$

Bucles

El caso más sencillo es el de un bucle **for**:

for j:= 1 **to** m **do** I

En el supuesto de que en I no se altera el índice `j`, esta instrucción tiene el siguiente coste:

$$m + \sum_{j=1}^{m} T_{I_j}(n)$$

donde la cantidad m representa las m veces que se incrementa `j` y la comprobación de si está entre los extremos inferior y superior.

En el caso de que el cuerpo del bucle consuma un tiempo fijo (independientemente del valor de `j`), la complejidad del bucle resulta ser $m(1 + T_I(n))$.

En los bucles **while** y **repeat** no hay una regla general, ya que no siempre se conoce el número de vueltas que da, y el coste de cada iteración no siempre es uniforme. Sin embargo, con frecuencia se puede acotar superiormente, acotando precisamente el número de vueltas y el coste de las mismas.

Subprogramas

El coste de ejecutar un subprograma no recursivo se deduce con las reglas descritas. Por el contrario, en caso de haber recursión, hay que detenerse a distinguir entre los casos básicos (los parámetros que no provocan nuevas llamadas recursivas) y los recurrentes (los que sí las producen). Considérese como ejemplo la versión recursiva de la función factorial (tomada del apartado 10.1):

```
function Fac (n: integer): integer;
  {PreC.:  n ≥ 0}
  {Dev.  n!}
begin
  if n = 0 then
    Fac:= 1
  else
    Fac:= n * Fac(n-1)
end;  {Fac}
```

Para calcular la complejidad $T_{\texttt{Fac}}(n)$ del algoritmo se debe tener en cuenta que, para el caso básico ($n = 0$), el coste es constante, $\Omega(1)$,

$$T_{\texttt{Fac}}(0) = 1$$

y para los recurrentes ($n > 0$), el coste es de una cantidad constante, $\Omega(1)$ más el de la llamada subsidiaria provocada:

$$T_{\texttt{Fac}}(n) = 1 + T_{\texttt{Fac}}(n-1)$$

En resumen,

$$T_{\texttt{Fac}}(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 + T_{\texttt{Fac}}(\texttt{n} - \texttt{1}) & \text{si } n > 0 \end{cases}$$

Así, para $n > 1$,

$$\begin{aligned} T_{\texttt{Fac}}(n) &= 1 + T_{\texttt{Fac}}(n-1) \\ &= 1 + 1 + T_{\texttt{Fac}}(n-2) \\ &= \ldots \\ &= n + T_{\texttt{Fac}}(0) \\ &= n + 1 \end{aligned}$$

y, por lo tanto, el coste es lineal, o sea, $T_{\texttt{Fac}}(n) \in \Omega(n)$.

18.3.2 Ejemplos

Una vez descrito cómo calcular la complejidad en tiempo de los programas en Pascal, se presentan algunos algoritmos a modo de ejemplo.

Producto de dos números enteros

Supóngase que se pretende calcular el producto de dos números naturales n y m sin utilizar la operación de multiplicación. Un primer nivel de diseño de un algoritmo para este problema es:

```
prod:= 0
repetir n veces
  repetir m veces
    prod:= prod + 1
```

Este diseño se implementa directamente en Pascal mediante la siguiente función:

```
function Producto(n, m: integer): integer;
  {PreC.: n,m ≥ 0}
  {Dev. n×m}
  var
    i,j,prod: integer;
begin
  prod:= 0;
  for i:= 1 to n do
    for j:= 1 to m do
      prod:= prod + 1;
  Producto:= prod
end; {Producto}
```

Como el primer bucle **for** se repite `n` veces y el segundo `m` veces, y puesto que el resto de las instrucciones son asignaciones (con tiempo de ejecución constante), se deduce que el algoritmo tiene una complejidad en tiempo $T(n, m) \in O(nm)$.

Para mejorar el algoritmo, se podría utilizar un único bucle, atendiendo al siguiente diseño:

```
prod:= 0
repetir n veces
  prod:= prod + m
```

Este diseño también se implementa fácilmente en Pascal mediante una función:

```
function Producto2(n, m: integer): integer;
  {PreC.: n,m ≥ 0}
  {Dev. n×m}
  var
    i,prod: integer;
begin
  prod:= 0;
  for i:= 1 to n do
    prod:= prod + m;
  Producto2:= prod
end; {Producto2}
```

Se obtiene así un código cuya complejidad es, claramente, $O(n)$.

Este algoritmo se puede mejorar ligeramente si se controla que el bucle se repita `n` veces si $n \leq m$, o `m` veces si $m \leq n$. La implementación de esta mejora se deja como ejercicio al lector.

Es posible conseguir una complejidad aún menor utilizando el algoritmo conocido con el nombre de *multiplicación a la rusa*.[6] El método consiste en multiplicar uno de los términos por dos mientras que el otro se divide por dos (división entera) hasta llegar a obtener un uno. El resultado se obtendrá sumando aquellos valores que se han multiplicado por dos tales que su correspondiente término dividido por dos sea un número impar. Por ejemplo, al realizar la *multiplicación a la rusa* de 25×11, se obtiene la siguiente tabla de ejecución:

25	11	$*$
50	5	$*$
100	2	
200	1	$*$

[6]Curiosamente, muchos autores dicen que este algoritmo es el que utilizaban los romanos para multiplicar.

obteniendo como resultado final $25 + 50 + 200 = 275$ debido a que sus términos divididos correspondientes ($11, 5$ y 1) son impares.

La justificación de por qué sumar sólo estos valores es la siguiente: si se tiene un producto de la forma $k \times (2l)$, un paso del algoritmo lo transforma en otro producto igual $(2k) \times l$; sin embargo, si el producto es de la forma $k \times (2l + 1)$, un paso del algoritmo lo transforma en $(2k) \times l$ (debido a que $(2l + 1)\ div\ 2 = l$), mientras que el verdadero resultado del producto es $k \times (2l + 1) = (2k) \times l + k$. Por lo tanto, se tendrá que sumar la cantidad perdida k y esto es lo que se hace siempre que el número que se divide es impar.

Dada la sencillez del algoritmo, se presenta directamente su implementación en Pascal:

```
function ProductoRuso(n,m:  integer):integer;
  {PreC.:  n,m ≥ 0}
  {Dev.  n×m}
  var
    prod:  integer;
begin
  prod:= 0;
  while m > 0 do begin
    if Odd(m) then
      prod:= prod + n;
    n:= n + n;  {n:= n * 2, sin usar *}
    m:= m div 2
  end;  {while}
  ProductoRuso:= prod
end; {ProductoRuso}
```

Como la variable `m` se divide por dos en cada vuelta del bucle, hasta llegar a 0, el algoritmo es de complejidad $O(\log\ m)$.

Ordenación de vectores

En el capítulo 15 se presentaron varios algoritmos para la ordenación de vectores, viendo cómo, intuitivamente, unos mejoraban a otros en su eficiencia en tiempo. En este apartado se estudia con detalle la complejidad de algunos de estos algoritmos[7] utilizando las técnicas expuestas en los apartados anteriores.

[7]Dado que no es necesario tener en cuenta todos los detalles de un algoritmo para su análisis, se ha optado en este apartado por estudiar la complejidad sobre el seudocódigo, llegando hasta el nivel menos refinado que permita su análisis. Los autores consideran que descender a niveles inferiores no es necesario, ya que los fragmentos que tardan un tiempo constante (o acotado por una constante) no importa qué detalles contengan.

Para empezar, se presenta el algoritmo de ordenación por intercambio directo (véase el apartado 15.2.3), que es fácil pero ineficiente, como se demostrará al estudiar su complejidad y compararla con la de los restantes. Como se explicó en dicho apartado, este algoritmo consiste en recorrer el array con dos bucles anidados dependientes. El primero recorre todos los elementos del vector, mientras que el segundo bucle va intercambiando los valores que están en orden decreciente. El boceto del algoritmo es el siguiente:

para `i` *entre* `1` *y* `n-1` *hacer*
  Desplazar el menor valor desde $\texttt{v}_{\texttt{n}}$ *hasta* $\texttt{v}_{\texttt{i}}$*, intercambiando*
  pares vecinos, si es necesario
Devolver `v` *ya ordenado*

Como siempre, para determinar la complejidad es necesario contar el número de veces que se ejecuta el cuerpo de los bucles, ya que las operaciones que intervienen en el algoritmo (asignaciones, comparaciones y acceso a elementos de un vector) se ejecutan en tiempo constante.

El cuerpo del bucle *Desplazar el menor valor... si es necesario* requiere `n-i` pasos en la vuelta `i`-ésima (uno para cada posible intercambio). Por lo tanto, el coste del algoritmo completo es[8]

$$\sum_{i=1}^{n}(n-i) = \frac{n(n-1)}{2}$$

y, en consecuencia, su complejidad es cuadrática ($O(n^2)$).

Analicemos ahora el algoritmo *Quick Sort* (véase el apartado 15.2.4) en el peor caso. A grandes trazos, el algoritmo es el siguiente:

si `v` *es de tamaño 1 entonces*
  `v` *ya está ordenado*
si no
  Dividir `v` *en dos bloques* `A` *y* `B`
  con todos los elementos de `A` *menores que los de* `B`
fin {`si`}
Ordenar `A` *y* `B` *usando Quick Sort*
Devolver `v` *ya ordenado como concatenación*
de las ordenaciones de `A` *y de* `B`

donde *Dividir* `v` *en dos bloques* `A` *y* `B` consiste en

[8]La suma que hay que calcular se corresponde con la suma de los términos de una progresión aritmética (véase el apartado 18.4).

Elegir un elemento `p` *(pivote) de* `v`
para cada elemento del vector hacer
  si el elemento `<` `p` *entonces*
    Colocar el elemento en `A`*, el subvector con los elementos de* `v`
    menores que `p`
  en otro caso
    Colocar el elemento en `B`*, el subvector con los elementos de* `v`
    mayores que `p`

Como se dijo en 15.2.4, se ha optado por elegir como pivote el primer elemento del vector.

En el peor caso (cuando el vector se encuentra ordenado decrecientemente), el algoritmo *Quick Sort* va a tener complejidad $O(n^2)$. La razón es que, en tal caso, el cuerpo del bucle *para cada elemento...* se ejecutará, en total, $(n-1)+(n-2)+\ldots+1$ veces, donde cada sumando proviene de cada una de las sucesivas ordenaciones recursivas del subvector `A`. Esto es así porque en cada llamada se ordena un solo elemento (el pivote), y por tanto dicho subvector tendrá sucesivamente longitud $(n-1),(n-2),\ldots,1$. Dicha suma, como se vio anteriormente es

$$\frac{n(n-1)}{2}$$

y por tanto el algoritmo tiene complejidad cuadrática. En resumen, la complejidad en el peor caso es la misma en el algoritmo anterior.

Ciertamente, en el capítulo 15 se presentó este último método como mejora en el tiempo de ejecución. Lo que ocurre es que esa mejora es la que se logra en el caso medio. Sin embargo, el correspondiente cálculo rebasa las pretensiones de este libro.

Para completar este apartado se presenta el análisis de la complejidad en tiempo de un tercer algoritmo de ordenación de vectores, concretamente el de ordenación por mezcla o *Merge Sort* (véase el apartado 15.2.5). El algoritmo es el que sigue:

si `v` *es de tamaño 1 entonces*
  `v` *ya está ordenado*
si no
  Dividir `v` *en dos subvectores* `A` *y* `B`
fin {`si`}
Ordenar `A` *y* `B` *usando Merge Sort*
Mezclar las ordenaciones de `A` *y* `B` *para generar el vector ordenado.*

En este caso, el paso *Dividir* `v` *en dos subvectores* `A` *y* `B` consiste en:

Asignar a `A` *el subvector* $[v_1, \ldots, v_{n \operatorname{div} 2}]$
Asignar a `B` *el subvector* $[v_{n \operatorname{div} 2+1}, \ldots, v_n]$

mientras que *Mezclar las ordenaciones de* `A` *y* `B` consiste en ir entremezclando adecuadamente las componentes ya ordenadas de `A` y de `B` para obtener el resultado buscado.

El análisis de la complejidad de este algoritmo no es complicado. Partiendo de que la operación de *Mezclar las ordenaciones de* `A` *y* `B` se ejecuta en un tiempo proporcional a n (la longitud del vector por ordenar), el coste en tiempo del algoritmo completo viene dado por la siguiente relación de recurrencia, donde k_i son cantidades constantes:

$$T(n) = \begin{cases} k_1 & \text{si } n = 1 \\ 2T(\frac{n}{2}) + k_2 n + k_3 & \text{si } n > 1 \end{cases}$$

En esta fórmula k_1 representa el coste del caso trivial (`v` de tamaño 1); $T(n/2)$ es el coste de cada llamada recursiva, y $k_2 n + k_3$ es el coste de mezclar los subvectores `A` y `B`, ya ordenados.

Esta ecuación se resuelve mediante sustituciones sucesivas cuando n es una potencia de 2 (es decir, existe j, tal que $n = 2^j$), de la siguiente forma:

$$\begin{aligned} T(n) &= 2(2T(\frac{n}{4}) + k_2\frac{n}{2} + k_3) + k_2 n + k_3 \\ &= 4T(\frac{n}{4}) + 2k_2 n + k_4 \\ &= 4(2T(\frac{n}{8}) + k_2\frac{n}{4} + k_3) + 2k_2 n + k_4 \\ &= \ldots \\ &= 2^j T(1) + jk_2 n + k_j \\ &= nk_1 + k_2 n \log_2 n + k_j \end{aligned}$$

Y en el caso en que n no sea una potencia de 2, siempre se verifica que existe $k > 0$ tal que $2^k < n < 2^{k+1}$, y, por tanto, se tiene que $T(n) \leq T(2^{k+1})$. En consecuencia, se puede afirmar que, en todo caso, $T(n) \in O(n \log_2 n)$.

Esta conclusión indica que *Merge Sort* es un algoritmo de ordenación con una complejidad en tiempo óptima[9] en el peor caso, aunque no es tan bueno en cuanto a la complejidad en espacio, ya que es necesario mantener dos copias del vector. Existen versiones mejoradas de este algoritmo que tienen menor coste en espacio, pero su estudio excede a las pretensiones de este texto.

[9]Hay que recordar que esta complejidad es óptima bajo la notación O-grande, esto es, salvo constantes de proporcionalidad.

18.3.3 Espacio de memoria empleado

Aunque el cálculo de la complejidad en espacio es similar al de la complejidad en tiempo, se rige por leyes distintas, como se comenta en este apartado.

En primer lugar, se debe tener en cuenta que la traducción del código fuente a código máquina depende del compilador y del computador, y que esto tiene una fuerte repercusión en la memoria. En consecuencia, y al igual que se razonó para la complejidad en tiempo, no es recomendable utilizar medidas absolutas sino relativas, como celdas de memoria (el espacio para almacenar, por ejemplo, un dato simple: un número o un carácter).

Si llamamos $S(\mathtt{n})$ al espacio relativo de memoria que el algoritmo ha utilizado al procesar una entrada de tamaño `n`, se definen los conceptos de $S_{\text{máx}}(\mathtt{n})$, $S_{\text{mín}}(\mathtt{n})$ y $S_{\text{med}}(\mathtt{n})$ para la complejidad en espacio del *peor caso, el mejor caso* y *caso medio*, de forma análoga a los conceptos respectivos de tiempo.

Con estos conceptos, y considerando, por ejemplo, que cada entero necesita una celda de memoria, el espacio necesario para la búsqueda secuencial ordenada es: `n` celdas para el vector, una celda para el elemento buscado y una celda para la variable `i`, es decir,

$$S(\mathtt{n}) = \mathtt{n} + 2$$

tanto en el peor caso como en mejor caso y en el caso medio.

De forma análoga se ve que el algoritmo de búsqueda binaria tiene como complejidad en espacio $S(\mathtt{n}) = \mathtt{n} + 4$.

Utilizando esta notación, podemos afirmar que la función `Sumatorio` del apartado 18.1 tiene, en su versión iterativa, una complejidad en espacio $S(\mathtt{n}) = 3$, debida al espacio ocupado por el parámetro `n` y las variables locales `i` y `suma`. La complejidad de la versión recursiva es $S(\mathtt{n}) = \mathtt{n} + 1$, puesto que cada una de las tablas de activación ocupa una celda para su parámetro local `n`.

Para el cálculo de la complejidad en espacio es preciso tener en cuenta algunos aspectos relacionados con el manejo de subprogramas:

- La llamada a un subprograma tiene asociado un coste en espacio, dado que se tiene que generar la tabla de activación (véase el apartado 10.2). Más concretamente:
  - Los parámetros por valor necesitan un espacio igual a su tamaño, al igual que los objetos (constantes y variables) locales.
  - Los parámetros por variable sólo necesitan una cantidad de espacio unitaria independientemente de su tamaño.
- Los algoritmos recursivos necesitan una cantidad de espacio dependiente de la "profundidad" de la recursión que determina el tamaño de la pila de tablas de activación (véase el apartado 17.2.3).

Por consiguiente, cuando un subprograma recursivo origine varias llamadas, sólo importará la llamada que provoque una mayor profundidad, pudiéndose despreciar las demás.

Es un error frecuente comparar la complejidad en espacio de los algoritmos recursivos con el número total de llamadas.

Ejemplo: sucesión de Fibonacci

La sucesión de los números de Fibonacci (véase el apartado 10.3.1) se puede hallar mediante la siguiente función recursiva:

```
function Fib(num: integer): integer;
  {PreC.:  num ≥ 0}
  {Dev.  fib_num}
begin
  if (num = 0) or (num = 1) then
    Fib:= 1
  else
    Fib:= Fib(num-1) + Fib(num-2)
end;  {Fib}
```

El coste en espacio, $S(n)$, del algoritmo descrito es proporcional a la profundidad del árbol de llamadas; es decir, $S(n) = 1$ en los casos triviales $n = 0$ y $n = 1$; en los no triviales ($n \geq 2$), `Fib(n)` origina dos llamadas subsidiarias, `Fib(n-1)` y `Fib(n-2)`, la primera de las cuales es más profunda. Por lo tanto, en estos casos,

$$S(n) = 1 + máx(S(n-1), S(n-2)) = 1 + S(n-1)$$

En resumidas cuentas, $S(n) = n$, lo que indica que esta función tiene un requerimiento de espacio lineal con respecto a su argumento n.

18.4 Útiles matemáticos

Ya se ha visto en los ejemplos anteriores que, cuando se trabaja con funciones o procedimientos recursivos, la complejidad en el tiempo $T(n)$ va a venir dada en función del valor de T en puntos menores que n. Por ello es útil saber cómo calcular términos generales de sucesiones en las que los términos se definen en función de los valores anteriores (sucesiones recurrentes). En este apéndice se tratan los casos más comunes que pueden surgir a la hora del cálculo de la complejidad en el tiempo de funciones o procedimientos recursivos.

18.4.1 Fórmulas con sumatorios

- Si x_n es una sucesión aritmética, esto es, $x_n = x_{n-1} + r$, entonces

$$x_1 + x_2 + \cdots + x_n = \frac{(x_1 + x_n)n}{2}$$

- $1 + x + x^2 + \cdots + x^{n-1} = \dfrac{1 - x^n}{1 - x}$.

- $\displaystyle\sum_{i=0}^{\infty} x^i = 1 + x + x^2 + \cdots = \frac{1}{1 - x}$, siempre que $|x| < 1$.

- $\displaystyle\sum_{i=0}^{\infty} \frac{x^i}{i!} = \mathrm{e}^x$.

- $\displaystyle\sum_{i=0}^{\infty} (-1)^i \frac{x^i}{i} = \log(x)$.

- Si cada $a_i \in \Delta(n^k)$, se tiene que $\displaystyle\sum_{i=1}^{n} a_i \in \Delta(n^{k+1})$.

18.4.2 Sucesiones de recurrencia lineales de primer orden

Son aquellas sucesiones en las que su término general viene dado en función del término anterior, es decir, $x_n = f(x_{n-1})$. En estas sucesiones es necesario conocer el valor de x_0.

Dependiendo de la forma f se consideran los siguientes casos:

- Si x_n es de la forma $x_n = cx_{n-1}$, se tiene que $x_n = c^n x_0$, ya que $x_n = cx_{n-1} = c^2 x_{n-2} = \ldots = c^n x_0$.

- Si x_n es de la forma $x_n = b_n x_{n-1}$ para $n \geq 1$, se tiene que $x_n = b_1 b_2 \ldots b_n x_0$.

- Si x_n es de la forma $x_n = b_n x_{n-1} + c_n$, realizando el cambio de variable $x_n = b_1 b_2 \ldots b_n y_n$ en la recurrencia de x_{n+1}, se obtiene:

$$b_1 b_2 \ldots b_{n+1} y_{n+1} = b_{n+1}(b_1 b_2 \ldots b_n y_n) + c_{n+1}$$

  lo que, operando, conduce a

$$x_n = (b_1 b_2 \ldots b_n) \times \left(x_0 + \sum_{i=1}^{n} d_i \right)$$

siendo $d_n = \dfrac{c_n}{(b_1 b_2 \dots b_n)}$.

Como ejemplo, se expone el caso particular que se obtiene cuando $b_n = b$ y $c_n = c$, es decir, $x_{n+1} = bx_n + c$. En este caso se realiza el cambio $x_n = b^n y_n$ y se tiene:

$$b^{n+1} y_{n+1} = b^{n+1} y_n + c \Rightarrow y_{n+1} = y_n + \frac{c}{b^{n+1}}$$

$$\Rightarrow y_n = y_0 + \sum_{i=1}^{n} \left(\frac{c}{b^i} \right) = y_0 + c \frac{1 - b^n}{1 - b}$$

lo que conduce a $x_n = x_0 + c \dfrac{1 - b^n}{(1 - b) b^n}$.

Estas recurrencias son de gran importancia por sí mismas, pero además, las recurrencias generadas por sustracción y por división se reducen a ellas.

Recurrencias generadas por sustracción

Existen algoritmos que dan lugar a recurrencias de la forma

$$x_n = Expr(n, x_{n-c})$$

conocido x_0. Mediante el cambio de variable $n = kc$ tenemos:

$$\begin{aligned} x_n &= Expr(n, x_{n-c}) \\ &= Expr(kc, x_{kc-c}) \\ x_{kc} &= Expr(kc, x_{(k-1)c}) \end{aligned}$$

Si ahora llamamos a la sucesión $x_{kc} = y_k$, tenemos:

$$\begin{aligned} y_0 &= x_0 \\ y_k &= x_{kc} \\ &= Expr(kc, y_{k-1}) \end{aligned}$$

que es lineal de primer orden. Una vez resuelta, se tiene que

$$x_n = y_{n/c}$$

Recurrencias generadas por división

Otros algoritmos dan lugar a recurrencias de la forma siguiente:

$$x_n = ...n...x_{n/c}$$

conocido x_0. Mediante el cambio de variable $n = c^k$ tenemos:

$$\begin{aligned} x_n &= Expr(n, x_{n/c}) \\ &= Expr(c^k, x_{c^k/c}) \\ x_{c^k} &= Expr(c^k, x_{c^{k-1}}) \end{aligned}$$

Si ahora llamamos a la sucesión $x_{c^k} = y_k$, tenemos:

$$\begin{aligned} y_0 &= x_0 \\ y_k &= x_{c^k} \\ &= Expr(c^k, y_{k-1}) \end{aligned}$$

que es lineal de primer orden. Una vez resuelta, se tiene que

$$x_n = y_{\log_c n}$$

Como ejemplo de este tipo de recurrencias, considérese el coste del algoritmo de ordenación por mezcla:

$$T(n) = \begin{cases} k_1 & \text{si } n = 1 \\ 2T(\frac{n}{2}) + k_2 n + k_3 & \text{si } n > 1 \end{cases}$$

Como su resolución sigue al pie de la letra el procedimiento descrito, se deja como ejercicio al lector. La solución puede compararse con la ofrecida en 18.3.2.

18.4.3 Sucesiones de recurrencia de orden superior

Son las sucesiones generadas por subprogramas recursivos con más de una llamada recursiva:[10]

$$x_n = f(x_{n_1}, x_{n_2}, \ldots, x_{n_k}, n)$$

La resolución exacta de este tipo de recurrencias sobrepasa las pretensiones de esta introducción a la complejidad algorítmica. Sin embargo, con frecuencia es posible y suficiente acotar dicho coste. En efecto, es frecuente que la sucesión x_i sea creciente y que entre los tamaños de las llamadas subsidiarias se puedan identificar el mínimo y el máximo y en general. Si llamamos $x_{mín}$ y $x_{máx}$ respectivamente a estos valores en la sucesión anterior, se tiene que

$$\left(\sum_{i=1}^{k} a_i\right) x_{mín} + f(n) \leq x_n \leq \left(\sum_{i=1}^{k} a_i\right) x_{máx} + f(n)$$

Esta desigualdad nos da siempre las acotaciones Θ y O y, cuando coincidan ambas, tendremos el orden Θ.

[10] Las distintas llamadas son x_{n_i}.

Ejemplo: sucesión de Fibonacci

En su definición recursiva usual, esta función tiene un coste dado por la siguiente relación de recurrencia:

$$\begin{array}{rcll} t_n & = & k_1 & \text{si } k = 0 \text{ o } k = 1 \\ t_n & = & x_{n-1} + x_{n-2} + k_2 & \text{si } k \geq 0 \end{array}$$

Como t es creciente, podemos acotarla entre f y g, así:

$$\begin{array}{rcll} f_n & = & k_1 & \text{si } k = 0 \text{ o } k = 1 \\ f_n & = & 2f_{n-2} + k_2 & \text{si } k \geq 0 \\ g_n & = & k_1 & \text{si } k = 0 \text{ o } k = 1 \\ g_n & = & 2g_{n-1} + k_2 & \text{si } k \geq 0 \end{array}$$

Estas dos relaciones de recurrencia son lineales, y se resuelven fácilmente:

$$f_n \in \Theta(2^{n/2}) g_n \in \Theta(2^n)$$

por lo que podemos concluir que la función analizada tiene un coste en tiempo exponencial $t_n = k^n$, para una constante k sin determinar, entre $\sqrt{2}$ y 2.

18.5 Ejercicios

1. Considere las siguientes funciones (dependientes de n) de cara a estudiar su comportamiento asintótico:

$$\begin{array}{c|||c|||c} n^2 + 10^3 n + 10^6 & n\sqrt{n} & \log_{10} n \\ (5n^2+3)(3n+2)(n+1) & \log\sqrt{n} & \sqrt{n} + \log\ n \\ 2^n & (\frac{1}{2})^n & 2^{1/n} \\ 3^n & 2^n n^2 & \log_e n \\ \frac{(n+1)(n^2-n+5)}{n(3+n^2)} & \frac{1}{n} & 1000 \\ \sum_{i=1}^{n} i & \sum_{i=1}^{n} n & \sum_{i=1}^{n}\sum_{j=1}^{i} n \end{array}$$

   Se pide lo siguiente:

   (a) Para cada una de ellas, busque una función sencilla que acote superiormente su comportamiento asintótico, (usando para ello la notación O mayúscula) procurando ajustarse lo más posible.

   (b) Clasifique las funciones anteriores según sean $O(2^n)$, $O(n^4)$, $O(\sqrt{n})$, $O(\log\ n)$ ó $O(1)$.

   (c) Agrupe las funciones del ejercicio anterior que sean del mismo orden Θ.

2. Compare las funciones siguientes por su orden de complejidad:

$$\log\ \log\ n \ \|\ \sqrt{\log\ n}\ \|\ (\log\ n)^2\ \|\ \sqrt[4]{n}$$

3. Calcule la complejidad en tiempo del siguiente algoritmo de sumar:

```
function Suma (m, n: integer): integer;
  {PreC.: n >= 0}
  {Dev. m + n }
  var
    i: integer;
begin
  for i:= 1 to n do
    m:= Succ (m)
end; {Suma}
```

4. Calcule ahora el coste del siguiente algoritmo de multiplicar,

```
function Producto (a, b: integer): integer;
  {PreC.: b >= 0}
  {Dev. a * b}
  var
    i, acumProd: integer;
begin
  acumProd:= 0;
  for i:= 1 to b do
    acumProd:= acumProd + a
end; {Producto}
```

en los tres casos siguientes:

(a) Tal como se ha descrito.

(b) Cambiando, en el cuerpo de la instrucción `for`, la expresión `acumProd + a` por la llamada `Suma(acumProd, a)`.

(c) Cambiando la expresión `acumProd + a` de antes por la llamada `Suma(a, acumProd)`.

5. Considerando los problemas siguientes, esboce algoritmos iterativos para resolverlos e indique su complejidad:

(a) Resolución de una ecuación de segundo grado.

(b) Cálculo del cociente y el resto de la división entera mediante restas sucesivas.

(c) Cálculo de un cero aproximado de una función mediante el método de bipartición.

(d) Suma de las cifras de un número entero positivo.

(e) Determinación de si un número es primo o no mediante el tanteo de divisores.

(f) Cálculo de $\sum_{i=1}^{n} \frac{i+1}{i!}$, diferenciando dos versiones:

i. Una, en la que cada vez se halla un término del sumatorio.

ii. Otra, donde cada factorial del denominador se halla actualizando el del término anterior.

(g) Ordenación de un array de n componentes por el método de intercambio directo:

```
for i:= 1 to n - 1 do
  for j:= n downto i + 1 do
    Comparar las componentes j-1 y j
    e intercambiar si es necesario
```

(h) Producto de dos matrices, una de $m \times k$ y otra de $k \times n$.

6. Para el problema de las Torres de Hanoi de tamaño n, calcule su complejidad exacta en tiempo y en espacio, así como su orden de complejidad.

   Calcule el tiempo necesario para transferir los 64 discos del problema tal como se planteó en su origen, a razón de un segundo por movimiento. De esta forma, podrá saber la fecha aproximada del fin del mundo según la leyenda.

7. Considerando los problemas siguientes, esboce algoritmos recursivos para resolverlos e indique su complejidad:

   (a) Cálculo del cociente de dos enteros positivos, donde la relación de recurrencia (en su caso) es la siguiente:

   ```
   Coc(dividendo, divisor) = 1 + Coc(dividendo - divisor,
                                     divisor)
   ```

   (b) Cálculo del máximo elemento de un array; así, si es unitario, el resultado es su único elemento; si no, se halla (recursivamente) el máximo de su mitad izquierda, luego el de su mitad derecha del mismo modo y luego se elige el mayor de estos dos números.

   (c) Cálculo de $\sum_{i=1}^{n} \frac{i+1}{i!}$, usando la relación de recurrencia siguiente, en su caso:

   $$\sum_{i=1}^{n} a_n = a_n + \sum_{i=1}^{n-1} a_n$$

   (d) Cálculo de los coeficientes binomiales $\binom{n}{k}$, distinguiendo las siguientes versiones:

      i. Iterativa, mediante el cálculo de $\frac{m!}{n!(m-n)!}$.
      ii. Iterativa, mediante el cálculo de $\frac{m(m-1)...(n+1)}{(m-n)!}$.
      iii. Recursiva, mediante la relación de recurrencia conocida.

   (e) Evaluación de un polinomio, conocido el valor de la variable `x`, en los siguientes casos:

      i. La lista de los coeficientes viene dada en un array, y se aplica la fórmula $\sum_i \text{coef}_i x^i$.
      ii. Los coeficientes están en una lista enlazada, y usamos la regla de Horner (véase el apartado 17.1.4).

   (f) Ordenación de un array de n componentes por el método de mezcla:

```
procedure MergeSort (var v: Vn(elem): iz, der);
  {Efecto: se ordena ascendentemente v[iz..der]}
begin
  if iz < der + 1 then begin
    MergeSort (la mitad izquierda de v);
    MergeSort (la mitad derecha de v);
    Mezclar las dos mitades de v
  end {if}
end; {MergeSort}
```

suponiendo que las mitades del vector son siempre de igual tamaño, para simplificar, y que el proceso de mezclarlas tiene un coste proporcional a su tamaño.

18.6 Referencias bibliográficas

La complejidad de algoritmos se estudia más o menos a fondo en casi todos los cursos de estructuras avanzadas de datos y de metodología de la programación. De las muchas referencias mencionables, omitiremos aquí las que tienen un contenido similar, y citaremos en cambio sólo algunas de las que pueden servir para ampliar lo expuesto aquí.

Para empezar, citamos [MSPF95], una suave introducción a los conceptos básicos de la complejidad. En el clásico libro de Brassard y Bratley [BB97] (especialmente en el capítulo 2) se pueden ampliar las técnicas estudiadas para resolver recurrencias. Otro libro interesante es [Wil89], que está completamente dedicado a la complejidad de algoritmos (incluyendo el estudio de algunos algoritmos conocidos de investigación operativa y de teoría de números útiles en criptografía), así como a la complejidad de problemas. En [BKR91], se estudia el tema y su aplicación en el desarrollo de programas y estructuras de datos con el objetivo de minimizar el coste. En este libro se introduce además el coste en el caso medio de algoritmos, así como el análisis de algoritmos paralelos.

Capítulo 19

Tipos abstractos de datos

A medida que se realizan programas más complejos, va aumentando simultáneamente la complejidad de los datos necesarios. Como se explicó en el capítulo 11, este aumento de complejidad puede afrontarse, en primera instancia, con las estructuras de datos proporcionadas por Pascal como son, por ejemplo, los arrays y los registros.

Cuando en un programa, o en una familia de ellos, el programador descubre una estructura de datos que se utiliza repetidamente, o que puede ser de utilidad para otros programas, es una buena norma de programación definir esa estructura como un nuevo tipo de datos e incluir variables de este tipo cada vez que sea necesario, siguiendo así un proceso similar al de la abstracción de procedimientos (véanse los apartados 8.1 y 9.3.1), mediante la cual se definen subprogramas que pueden ser reutilizados. En este proceso, denominado *abstracción de datos*, el programador debe despreocuparse de los detalles menores, concentrándose en las operaciones globales del tipo de datos. Esto es, en términos generales, lo que se persigue con los *tipos abstractos de datos*.

19.1 Introducción

El objetivo que se persigue en este apartado es la definición de un conjunto de objetos con una serie de operaciones para su manipulación. Para resolverlo, se utiliza la técnica de la *abstracción de datos*, que permite tratar estas definiciones de tipos de una forma ordenada, mantenible, reutilizable y coherente.

La abstracción de datos pone a disposición del programador-usuario[1] nuevos tipos de datos con sus correspondientes operaciones de una forma totalmente independiente de la representación de los objetos del tipo y de la implementación de las operaciones; de ahí el nombre *tipos abstractos de datos*.

Esta técnica es llamada *abstracción*, porque aplica un proceso consistente en ignorar ciertas características de los tipos de datos por ser irrelevantes para el problema que se intenta resolver. Esas características ignoradas son aquéllas relativas a *cómo* se implementan los datos, centrándose toda la atención en *qué* se puede hacer con ellos. Esto es, las propiedades de un tipo abstracto de datos vienen dadas implícitamente por su definición y no por una representación o implementación particular.

Obsérvese que los tipos de datos básicos de Pascal (véase el capítulo 3) son *abstractos* en el siguiente sentido: el programador puede disponer de, por ejemplo, los enteros y sus operaciones (representados por el tipo `integer`), ignorando la representación concreta (complemento restringido o auténtico, o cualquiera de las explicadas en el apartado 2.2 del tomo I) escogida para éstos.

En cambio, con los tipos definidos por el programador (por ejemplo, las colas presentadas en el apartado 17.3), los detalles de la implementación están a la vista, con los siguientes inconvenientes:

- El programador tiene que trabajar con la representación de un objeto en lugar de tratar con el objeto directamente.
- El programador podría usar el objeto de modo inconsistente si manipula inadecuadamente la representación de éste.

En resumen, desde el punto de vista del programador-usuario se puede afirmar que la introducción de los tipos abstractos de datos suponen un aumento de nivel en la programación, pues bastará con que éste conozca el *qué*, despreocupándose de las características irrelevantes (el *cómo*) para la resolución del problema. Por otra parte, la tarea del programador que implementa el tipo consistirá en escoger la representación concreta que considere más adecuada y ocultar los detalles de ésta en mayor o menor nivel, dependiendo del lenguaje utilizado.

[1] Designaremos con este término al programador que utiliza porciones de código puestas a su disposición por otros programadores (o por él mismo, pero de forma independiente).

19.2 Un ejemplo completo

Es conveniente concretar todas estas ideas mediante un ejemplo: como se vio en el apartado 11.3, la representación de conjuntos en Pascal tiene una fuerte limitación en cuanto al valor máximo del cardinal de los conjuntos representados (por ejemplo, en Turbo Pascal el cardinal máximo de un conjunto es 256). En el siguiente ejemplo, se pretende representar conjuntos sin esta restricción. Posteriormente, se utilizarán estos conjuntos ilimitados en un programa que escriba los números primos menores que uno dado por el usuario, usando el conocido método de la *criba de Eratóstenes* (véase el apartado 11.3.3). La idea de la representación es disponer de un conjunto inicial con todos los enteros positivos menores que el valor introducido por el usuario e ir eliminando del conjunto aquellos números que se vaya sabiendo que no son primos. De acuerdo con esta descripción, una primera etapa de diseño del programa podría ser:

Leer `cota` $\in \mathbb{N}$
Generar el `conjunto` *inicial,* {*2, ...,* `cota`}
Eliminar los números no primos del `conjunto`
Escribir los números del `conjunto`

Detallando un poco más cada una de esas acciones, se tiene: *Generar el conjunto inicial* se puede desarrollar así:

Crear un conjunto vacío `primos`
Añadir a `primos` *los naturales de 2 a* `cota`

Para *Eliminar los números no primos del* `conjunto`, basta con lo siguiente:

para cada elemento `e`$\in$`conjunto`*, entre* `2` *y* $\sqrt{\texttt{cota}}$
  Eliminar del `conjunto` *todos los múltiplos de* `e`

En un nivel de refinamiento inferior, se puede conseguir *Eliminar del conjunto todos los múltiplos de* `e` de la siguiente forma:

```
coeficiente:= 2;
repetir
  Eliminar e * coeficiente del conjunto
  coeficiente:= coeficiente + 1
hasta que e * coeficiente sea mayor que cota
```

Finalmente, *Escribir los números del* `conjunto` no presenta problemas y se puede hacer con un simple recorrido de los elementos del `conjunto`.

19.2.1 Desarrollo de programas con tipos concretos de datos

Una vez detallado este nivel de refinamiento, se tiene que tomar una decisión sobre el modo de representar los conjuntos. Las posibilidades son múltiples: con el tipo **set** en Pascal, con listas enlazadas, etc.

En este apartado se presenta una implementación del diseño anterior, representando un conjunto de enteros en una lista enlazada con cabecera, con los elementos ordenados ascendentemente y sin repeticiones. Desde el nivel de refinamiento alcanzado en el apartado anterior se puede pasar directamente a la implementación:

```
Program CribaEratostenes (input, output);
  {PreC.:  input = [un entero, >=2]}
  type
    tConjunto = ^tNodoEnt;
    tNodoEnt = record
      elem:  integer;
      sig:  tConjunto
    end; {tNodoEnt}
  var
    cota, e, coef:  integer;
    conjunto, aux, puntPrimo, auxElim:  tConjunto;
begin
    {Leer cota:}
  Write('Cota:  ');
  ReadLn(cota);
    {Generar el conjunto inicial, [2, ..., cota]:}
  New(conjunto);
  aux:= conjunto;
  for e:= 2 to cota do begin
    New(aux^.sig);
    aux:= aux^.sig;
    aux^.elem:= e
  end; {for i}
  aux^.sig:= nil;
    {Eliminar los números no primos del conjunto:}
  puntPrimo:= conjunto^.sig;
  repeat
    e:= puntPrimo^.elem;
    coef:= 2; aux:= puntPrimo;
    while (e * coef <= cota) and (aux^.sig <> nil) do begin
      if aux^.sig^.elem < coef * e then
        aux:= aux^.sig
      else if aux^.sig^.elem = coef * e then begin
        auxElim:= aux^.sig^.sig;
        Dispose(aux^.sig);
```

```
          aux^.sig:= auxElim;
          coef:= coef + 1
        end {else if}
        else if aux^.sig^.elem > coef * e then
          coef:= coef + 1
      end; {while}
      puntPrimo:= puntPrimo^.sig
    until (e >= Sqrt(cota)) or (puntPrimo = nil);
      {Escribir los números del conjunto:}
    aux:= conjunto^.sig;
    while aux <> nil do begin
      Write(aux^.elem:4);
      aux:= aux^.sig
    end; {while}
    WriteLn
  end.  {CribaEratostenes}
```

Queda claro que a partir del momento en que se ha adoptado esta representación, quedan mezclados los detalles relativos al algoritmo (Criba de Eratóstenes) con los relativos a la representación (lista enlazada ...) y manipulación de los conjuntos de enteros. Este enfoque acarrea una serie de inconvenientes, como son:

- El código obtenido es complejo, y, en consecuencia, se dificulta su corrección y verificación.
- El mantenimiento del programa es innecesariamente costoso: si, por ejemplo, se decidiese cambiar la estructura dinámica lineal por una de manejo más eficiente, como podrían ser los árboles binarios de búsqueda, se debería rehacer la totalidad del programa. Dicho de otro modo, es muy difícil aislar cambios o correcciones.
- En el caso en que se necesitasen conjuntos sin restricciones de cardinal en otros programas, sería necesario volver a implementar en éstos todas las tareas necesarias para su manipulación. En otras palabras, no hay posibilidad de reutilizar código.

Estos inconvenientes son los que se pretende superar con los tipos abstractos de datos.

19.2.2 Desarrollo de programas con tipos abstractos de datos

Una forma de solucionar los problemas enunciados anteriormente es el empleo de la abstracción de datos. Así, se definirá un tipo abstracto de datos, entendido

informalmente como una colección de objetos con un conjunto de operaciones definidas sobre estos objetos. Se tendrá siempre en cuenta la filosofía de la abstracción de datos: todas las características del tipo abstracto vienen dadas por su definición y no por su implementación (de la que es totalmente independiente).

En el ejemplo anterior, es evidente que la aplicación de la abstracción de datos conducirá a un tipo abstracto:[2]

```
type
  tConj = Abstracto
```

cuyos objetos son precisamente conjuntos con un número arbitrario de elementos enteros que se podrán manipular con las siguientes operaciones:[3]

```
procedure CrearConj (var conj: tConj);
  {Efecto: conj:= ∅}

procedure AnnadirElemConj (elem: integer; var conj: tConj);
  {Efecto: conj:= conj ∪ [elem]}

procedure QuitarElemConj (elem: integer; var conj: tConj);
  {Efecto: conj:= conj \ [elem]}

function Pertenece (elem: integer; conj: tConj): boolean;
  {Dev. True (si elem ∈ conj) o False (en otro caso)}

procedure EscribirConj (conj: tConj);
  {Efecto: escribe en el output los elementos de conj}

function EstaVacioConj(conj: tConj): boolean;
  {Dev. True (si conj = ∅) o False (en otro caso)}
```

Como se dijo anteriormente, una característica esencial de los tipos abstractos de datos es su independencia de la implementación. En este momento, y sin saber nada en absoluto acerca de la forma en que está implementado (o en que se va a implementar) el tipo `tConj`, se puede utilizar de una forma abstracta, siendo más que suficiente la información proporcionada por la especificación de las operaciones del tipo. Además, se podrá comprobar inmediatamente cómo el código obtenido es más claro, fácil de mantener y verificar. La nueva versión del programa `CribaEratostenes` es la siguiente:

[2] Acéptese esta notación provisional, que se detalla en el apartado 19.2.3.

[3] En los casos en los que se ha considerado conveniente, se ha sustituido la postcondición por una descripción algo menos formal del efecto del subprograma.

```
Program CribaEratostenes (input, output);
  {PreC.:  input = [un entero, >=2]}
  type
    tConj = Abstracto;
  var
    cota, e, coef:  integer;
    conjunto :  tConj;
begin
    Write('Cota:  '); ReadLn(cota);
    CrearConj(conjunto);
    for e:= 2 to cota do
      AnnadirElemConj(e, conjunto);
    for e:= 2 to Trunc(SqRt(cota)) do
      if Pertenece (e, conjunto) then begin
          coef:= 2;
          repeat
            QuitarElemConj(e * coef, conjunto);
            coef:= coef + 1
          until e * coef > cota
      end; {if}
    EscribirConj(conjunto)
end.  {CribaEratostenes}
```

En el ejemplo se puede observar la esencia de la abstracción de datos: el programador-usuario del tipo abstracto de datos se puede olvidar completamente de *cómo* está implementado o de la representación del tipo, y únicamente estará interesado en qué se puede hacer con el tipo de datos `tConj` que utiliza de una forma completamente abstracta.

Además, el lector puede observar cómo se llega a una nueva distribución (mucho más clara) del código del programa: las operaciones sobre el tipo `tConj` dejan de formar parte del programa y pasan a incorporarse al código propio del tipo abstracto.

Las operaciones del tipo `tConj` escogidas son típicas de muchos tipos abstractos de datos, y se suelen agrupar en las siguientes categorías:

Operaciones de creación: Son aquéllas que permiten obtener nuevos objetos del tipo, como sucede en `CrearConj`. Estas últimas son conocidas también como operaciones constructoras primitivas. Entre éstas suele incluirse una operación de lectura de elementos del tipo abstracto de datos.

Operaciones de consulta: Realizan funciones que, tomando como argumento un objeto del tipo abstracto, devuelven un valor de otro tipo, como hacen `Pertenece` o `EstaVacioConj`, por poner un caso. Usualmente implementan tareas que se ejecutan con relativa frecuencia. Entre éstas se suele incluir

una operación que escriba objetos del tipo abstracto de datos, que en el ejemplo sería un procedimiento `EscribirConj`.

Operaciones de modificación: Permiten, como su propio nombre indica, modificar un objeto del tipo abstracto de datos, como, por ejemplo, las operaciones `Annadir` y `Eliminar`.

Operaciones propias del tipo: Son operaciones características de los objetos abstraídos en el tipo de datos, como serían en el ejemplo operaciones para calcular la unión, intersección o diferencia de conjuntos.

Es conveniente destacar que esta clasificación de las operaciones no es excluyente: en algún tipo abstracto de datos puede ser necesaria una operación que pertenezca a dos clases, como, por ejemplo, una operación que efectúe una consulta y una modificación al mismo tiempo. No obstante, tal operación no es adecuada desde el punto de vista de la cohesión, un criterio de calidad de *software* aplicable a la programación con subprogramas, que nos recomienda dividir tal operación en dos, una que haga la consulta y otra la modificación.

A modo de resumen, se puede decir que la abstracción de datos, considerada como método de programación, consiste en el desarrollo de las siguientes etapas:

1. Reconocer los objetos candidatos a elementos del nuevo tipo de datos.
2. Identificar las operaciones del tipo de datos.
3. Especificar las operaciones de forma precisa.
4. Seleccionar una buena implementación.

Se han mostrado las tres primeras etapas tomando como guía el ejemplo de los conjuntos de enteros. A continuación se detalla cómo abordar la cuarta y última.

19.2.3 Desarrollo de tipos abstractos de datos

En la última versión del ejemplo de la criba de Eratóstenes se ha utilizado el tipo abstracto `tConj` dejándolo sin desarrollar, únicamente incluyendo la palabra *abstracto*. Pero, obviamente, por mucho que las características de un tipo abstracto de datos sean independientes de la implementación, no se puede olvidar ésta.

Lo más adecuado para implementar un tipo abstracto de datos es recurrir a mecanismos que permitan *encapsular* el tipo de datos y sus operaciones, y *ocultar* la información al programador-usuario. Estas dos características son esenciales

para llevar a cabo efectivamente la abstracción de datos. A tal fin, Turbo Pascal dispone de las *unidades* (véase el apartado B.11).

De acuerdo con esto, se da a continuación una posible implementación[4] del tipo abstracto de datos `tConj`, a base de listas de enteros, ordenadas ascendentemente, enlazadas con punteros:[5]

```
unit conjEnt;
  {Implementación mediante listas enlazadas, con cabecera,
   ordenadas ascendentemente y sin repeticiones}

interface
  type
    tElem = integer;
      {Requisitos:  definidas las relaciones '=' (equiv.)  y
        '>' (orden total)}
    tConj = ^tNodoLista;
    tNodoLista = record
      info:  tElem;
      sig:  tConj
    end; {tNodoLista}

  procedure CrearConj(var conj:  tConj);
    {Efecto:  conj:= ∅}

  procedure DestruirConj(var conj:  tConj);
    {Cuidado:  se perderá toda su información}
    {Efecto:  conj:= ?  (ni siquiera queda
      vacío:  comportamiento impredecible)}

  procedure AnnadirElemConj (elem:  tElem; var conj:  tConj);
    {Efecto:  conj:= conj ∪ [elem]}

  procedure QuitarElemConj (elem:  tElem; var conj:  tConj);
    {Efecto:  conj:= conj \ [elem]}

  function Pertenece (elem:  tElem; conj:  tConj):  boolean;
    {Dev.  True (si elem ∈ conj) o False (en otro caso)}
```

[4]Se ha optado por no detallar todas las etapas del diseño descendente de las operaciones de los tipos abstractos, por ser éstas sencillas y para no extender demasiado el texto.

Por otra parte, es conveniente que los identificadores de unidades coincidan con el nombre del archivo donde se almacenan una vez compiladas (de ahí la elección de identificadores de, a lo sumo, ocho caracteres).

[5]Obsérvese que se ha incluido una operación destructora, necesaria en la práctica, debido a que el manejo de listas enlazadas ocasiona un gasto de memoria y ésta debe liberarse cuando un conjunto deje de ser necesario.

```
  procedure EscribirConj (conj:  tConj);
    {Efecto:  escribe en el output los elementos de conj}

  function EstaVacioConj(conj:  tConj):  boolean;
    {Dev.  True (si conj = ∅) o False (en otro caso)}

implementation

  {Representación mediante listas enlazadas, con cabecera,
   ordenadas ascendentemente y sin repeticiones}

  procedure CrearConj (var conj:  tConj);
  begin
    New(conj);
    conj^.sig:= nil
  end; {CrearConj}

  procedure DestruirConj (var conj:  tConj);
    var
      listaAux:  tConj;
  begin
    while conj <> nil do begin
      listaAux:= conj;
      conj:= conj^.sig;
      Dispose(listaAux)
    end {while}
  end; {DestruirConj}

  procedure AnnadirElemConj (elem:  tElem; var conj:  tConj);
    var
      parar, {indica el fin de la búsqueda, en la lista}
      insertar:  boolean; {por si elem ya está en la lista}
      auxBuscar, auxInsertar:  tConj;
  begin
    auxBuscar:= conj;
    parar:= False;
    repeat
      if auxBuscar^.sig = nil then begin
        parar:= True;
        insertar:= True
      end {then}
      else if auxBuscar^.sig^.info >= elem then begin
        parar:= True;
        insertar:= auxBuscar^.sig^.info > elem
      end {then}

      else
```

```
      auxBuscar:= auxBuscar^.sig
  until parar;
  if insertar then begin
    auxInsertar:= auxBuscar^.sig;
    New(auxBuscar^.sig);
    auxBuscar^.sig^.info:= elem;
    auxBuscar^.sig^.sig:= auxInsertar
  end {if}
end; {AnnadirElemConj}

procedure QuitarElemConj (elem:  tElem; var conj:  tConj);
  var
    parar, {indica el fin de la búsqueda, en la lista}
    quitar:  boolean; {por si el elem no está en la lista}
    auxBuscar, auxQuitar:  tConj;
begin
  auxBuscar:= conj;
  parar:= False;
  repeat
    if auxBuscar^.sig = nil then begin
      parar:= True;
      quitar:= False
    end {then}
    else if auxBuscar^.sig^.info >= elem then begin
      parar:= True;
      quitar:= auxBuscar^.sig^.info = elem
    end
    else
      auxBuscar:= auxBuscar^.sig
  until parar;
  if quitar then begin
    auxQuitar:= auxBuscar^.sig^.sig;
    Dispose(auxBuscar^.sig);
    auxBuscar^.sig:= auxQuitar
  end {if}
end; {QuitarElemConj}

function Pertenece (elem:  tElem; conj:  tConj):  boolean;
  var
    parar, {indica el fin de la búsqueda, en la lista}
    esta:  boolean; {indica si elem ya está en la lista}
    auxBuscar:  tConj;
begin
  auxBuscar:= conj^.sig;
  parar:= False;

  repeat
```

```
      if auxBuscar = nil then begin
        parar:= True;
        esta:= False
      end
      else if auxBuscar^.info = elem then begin
        parar:= True;
        esta:= True
      end
      else if auxBuscar^.info > elem then begin
        parar:= True;
        esta:= False
      end
      else
        auxBuscar:= auxBuscar^.sig
    until parar;
    pertenece:= esta
  end; {Pertenece}

  procedure EscribirConj (conj:  tConj);
    var
      puntAux:  tConj;
  begin
    if EstaVacioConj(conj) then
      WriteLn('[]')
    else begin
      puntAux:= conj^.sig;
      Write('[', puntAux^.info);
      while not EstaVacioConj(puntAux) do begin
        puntAux:= puntAux^.sig;
        Write(', ', puntAux^.info)
      end; {while}
      WriteLn(']')
    end {else}
  end; {EscribirConj}

  function EstaVacioConj (conj:  tConj):  boolean;
  begin
    EstaVacioConj:= conj^.sig = nil
  end; {EstaVacioConj}
end. {conjEnt}
```

Para finalizar, se recuerda que en el programa `CribaEratostenes` se había dejado incompleta la declaración de tipos, así, únicamente

```
  type
    tConj =  Abstracto;
```

Con los elementos de que se dispone ahora, esta declaración se sustituirá por la siguiente:

```
uses conjEnt;
```

que hace que el tipo `tConj` declarado en la unidad esté disponible para su empleo en el programa.

Una implementación en Pascal estándar obligaría a prescindir de las unidades, obligando a incluir todas las declaraciones del tipo y de los subprogramas correspondientes a las operaciones del tipo abstracto `tConj` en todo programa en que se utilice. Con ello se perderían gran parte de las ventajas aportadas por la abstracción de datos, como, por ejemplo, la *encapsulación*, la *ocultación* de información y el aislamiento de los cambios.

La representación escogida para la implementación no es la más eficiente. En efecto, las operaciones de inserción, eliminación y consulta requieren, en el peor caso, el recorrido del conjunto completo.

Una posibilidad más interesante consiste en representar los conjuntos mediante arboles de búsqueda (véase el apartado 17.4.2) en vez de usar listas. No es necesario reparar ahora en los pormenores de esta estructura de datos; en este momento, nos basta con saber que las operaciones de modificación y consulta son ahora mucho más eficientes.

Pues bien, si ahora decidiésemos mejorar la eficiencia de nuestra implementación, cambiando las listas por árboles, no tenemos que modificar las operaciones de la interfaz, sino tan sólo su desarrollo posterior en la sección **implementation**. De este modo, se puede apreciar una de las ventajas de la abstracción de datos:

- El programador-usuario no tiene que pensar en ningún momento en la representación del tipo abstracto de datos, sino únicamente en su especificación.
- Los cambios, correcciones o mejoras introducidas en la implementación del tipo abstracto de datos repercuten en el menor ámbito posible, cual es la unidad en que se incluye su representación. En nuestro ejemplo, el cambio de la implementación basada en listas por la basada en árboles no afecta ni a una sola línea del programa `Criba`: el programador-usuario sólo se sorprenderá con una mayor eficiencia de su programa tras dicho cambio. Obsérvese el efecto que hubiera tenido el cambio de representación en la primera versión de `Criba`, y compare.

Una vez presentado este ejemplo y con él las ideas generales de la abstracción de datos, se dan en el siguiente apartado nociones metodológicas generales acerca de esta técnica.

19.3 Metodología de la programación de tipos abstractos de datos

En los apartados anteriores se han introducido los tipos abstractos de datos de un modo paulatino, práctico e informal. Es necesario, y es lo que se hace en este apartado, precisar las ideas generales introducidas y presentar los aspectos necesarios para la correcta utilización de los tipos abstractos de datos como método de programación.

Siguiendo las ideas de J. Martin [Mar86], se puede definir un tipo abstracto de datos como un sistema con tres componentes:

1. Un conjunto de objetos.

2. Un conjunto de descripciones sintácticas de operaciones.

3. Una descripción semántica, esto es, un conjunto suficientemente completo de relaciones que especifiquen el funcionamiento de las operaciones.

Se observa que, mientras en el apartado anterior se describían los tipos abstractos de datos como un conjunto de objetos y una colección de operaciones sobre ellos, ahora se subraya la descripción de la sintaxis y la semántica de esas operaciones.

Esto es necesario dado que la esencia de los tipos abstractos de datos es que sus propiedades vienen descritas por su especificación, y, por tanto, es necesario tratar ésta adecuadamente.

Además, este tratamiento riguroso de las especificaciones de los tipos abstractos de datos permite ahondar en la filosofía expuesta al comienzo del tema: separar *qué* hace el tipo abstracto de datos (lo cual viene dado por la especificación) de *cómo* lo lleva a cabo (lo que se da en su implementación). Estos dos aspectos se repasan brevemente en los dos siguientes apartados.

19.3.1 Especificación de tipos abstractos de datos

A la hora de afrontar la especificación de un tipo abstracto de datos se dispone de diversos lenguajes variando en su nivel de formalidad. En un extremo, se tiene el lenguaje natural, en nuestro caso el español. Las especificaciones así expresadas presentan importantes inconvenientes, entre los que destacan su ambigüedad (que puede llegar a inutilizar la especificación, por prestarse a interpretaciones incorrectas y/o no deseadas), y la dificultad para comprobar su corrección y "completitud".

Ante estos problemas, es necesario recurrir a lenguajes más precisos, como puede ser el lenguaje matemático, que posee las ventajas de su precisión, concisión y universalidad (aunque en su contra se podría argumentar su dificultad de uso, lo cierto es que en los niveles tratados en este libro no es preciso un fuerte aparato matemático). Por ejemplo, algunas de las operaciones del tipo abstracto `tConj` se han especificado como sigue:

```
procedure CrearConj(var conj: tConj);
  {Efecto: conj:= ∅}

procedure AnnadirElemConj(elem: integer; var conj: tConj);
  {Efecto: conj:= conj ∪ [elem]}

function Pertenece (elem: tElem; conj: tConj): boolean;
  {Dev. True (si elem ∈ conj) o False (en otro caso)}

function EstaVacioConj(conj: tConj): boolean;
  {Dev. True (si conj = ∅) o False (en otro caso)}
```

Obsérvese que la sintaxis de su uso viene dada por los encabezamientos de las operaciones y que su semántica se da en el comentario que les sigue. Las especificaciones de esta forma proporcionan un modelo más o menos formal que describe el comportamiento de las operaciones sin ningún tipo de ambigüedad. Además, se satisfacen las dos propiedades que deben cumplir las especificaciones: precisión y brevedad.

Por último, se debe resaltar la importancia de la especificación como medio de comunicación entre el programador-diseñador del tipo abstracto de datos, el implementador y el programador-usuario: como se ha repetido anteriormente, la información pública del tipo abstracto de datos debe ser únicamente su especificación. Así, una especificación incorrecta o incompleta impedirá a los implementadores programar adecuadamente el tipo abstracto de datos, mientras que los programadores-usuarios serán incapaces de predecir correctamente el comportamiento de las operaciones del tipo, lo que producirá errores al integrar el tipo abstracto de datos en sus programas.

19.3.2 Implementación de tipos abstractos de datos

Como se indicó anteriormente, la tercera etapa de la abstracción de datos es la implementación del tipo de acuerdo con la especificación elaborada en la etapa anterior.

La idea de partida que se ha de tomar en esta tarea es bien clara: escoger una representación que permita implementar las operaciones del tipo abstracto de datos simple y eficientemente, respetando, por supuesto, las eventuales restricciones existentes.

Ahora bien, en el momento de llevar a la práctica esta idea, se ha de tener presente que el verdadero sentido de la abstracción de datos viene dado por la separación del *qué* y del *cómo*, de la especificación y de la implementación, y de la ocultación al programador-usuario de la información referente a esta última.

De acuerdo con esto, y como ya se adelantó en el apartado 19.2.3, resulta imprescindible disponer en el lenguaje de programación empleado de herramientas que permitan la implementación separada de los programas y los datos. Estas herramientas se denominan *unidades*, *módulos* o *paquetes* en algunos de los lenguajes de programación imperativa más extendidos. Pero estas herramientas no están disponibles en Pascal estándar, por lo que es imposible realizar la abstracción de datos de una forma completa[6] en este lenguaje, como se advirtió en el apartado anterior. Sin embargo, en Turbo Pascal sí que se dispone de la posibilidad de compilación separada mediante el empleo de *unidades* (véase el apartado B.11), y esto es suficiente para llevar a la práctica la abstracción de datos.

En otros términos, las unidades de Turbo Pascal posibilitan una total "encapsulación" de los datos, al incluir la definición y la implementación del tipo abstracto en la misma unidad. Ésta es una característica positiva, al aumentar la modularidad de los programas y facilitar el aislamiento de los cambios.

No obstante, las unidades permiten ocultar sólo parcialmente las características del tipo abstracto de datos dadas por la representación escogida. Es preciso recalcar la parcialidad de la ocultación de información, ya que es necesario incluir la representación concreta en la declaración del tipo abstracto (en el ejemplo `tConj = ^tNodoLista;` o `tConj =` *árbol de búsqueda*) contenida en la sección de interfaz de la unidad, y, por tanto, el programador-usuario conocerá parte de los detalles de la representación.[7]

Para finalizar, se puede decir que la implementación típica de un tipo abstracto de datos en Turbo Pascal, siguiendo las ideas de Collins y McMillan [CM], tiene la siguiente estructura:

```
unit TipoAbstractoDeDatos;
```

[6] En Pascal estándar habría que incluir la implementación del tipo de datos y de sus operaciones en cada programa que lo necesite, con lo cual no existe ninguna separación entre la especificación y la implementación, que estará totalmente visible al programador-usuario.

[7] En otros lenguajes, como, por ejemplo, Modula2, se puede conseguir la ocultación total de información mediante el empleo de los llamados *tipos opacos*.

interface
uses *Otras unidades necesarias*;

Declaraciones de constantes, tipos, y variables necesarios para definir el tipo abstracto

Encabezamientos de las operaciones del tipo abstracto de datos

implementation
uses *Otras unidades necesarias*;

Información privada, incluyendo las implementaciones de las operaciones del tipo abstracto de datos y de los tipos participantes, así como las constantes, tipos y variables necesarios para definir éstos, y las operaciones privadas

begin
Código de iniciación, si es preciso

end. {TipoAbstractoDeDatos}

19.3.3 Corrección de tipos abstractos de datos

La verificación de un tipo abstracto de datos debe hacerse a dos niveles:

En primer lugar, se debe estudiar si la implementación de las diferentes operaciones satisface la especificación, bien mediante una verificación *a posteriori*, o bien a través de una derivación correcta de programas (en los términos presentados en el apartado 5.4) partiendo de la especificación y desarrollando las operaciones de acuerdo con ésta. Siguiendo esta segunda opción, por ejemplo, para la operación `AnnadirElemConj`, se partiría de la especificación

{`Efecto: conj:= conj` $\cup$ [`elem`]}

y se llegaría al código

```
procedure AnnadirElemConj (elem: tElem; var conj: tConj);
  var
    parar, {indica el fin de la búsqueda, en la lista}
    insertar: boolean; {por si elem ya está en la lista}
    auxBuscar, auxInsertar: tConj;
begin
  auxBuscar:= conj;
  parar:= False;
  repeat
    if auxBuscar^.sig = nil then begin
```

```
        parar:= True;
        insertar:= True
      end
      else if auxBuscar^.sig^.info >= elem then begin
        parar:= True;
        insertar:= auxBuscar^.sig^.info > elem
      end
      else
        auxBuscar:= auxBuscar^.sig
    until parar;
    if insertar then begin
      auxInsertar:= auxBuscar^.sig;
      New(auxBuscar^.sig);
      auxBuscar^.sig^.info:= elem;
      auxBuscar^.sig^.sig:= auxInsertar
    end {if}
end; {AnnadirElemConj}
```

que verifica la especificación, como es fácil comprobar examinando el código. En efecto, la implementación propuesta para `AnnadirElemConj`, hace que se agregue `elem` a la representación de `conj`. Más detalladamente, en el caso del que `elem` no pertenezca al conjunto, se asigna el valor `True` a la variable `insertar`, provocando la ejecución de la rama **then** de la instrucción **if** `insertar...`, que añade un nuevo nodo (cuya información es `elem`) en la lista que representa a `conj`. Esto demuestra informalmente que `AnnadirElemConj` consigue el efecto descrito en la especificación.

Por otra parte, en los tipos abstractos de datos, se debe estudiar la corrección en una segunda dirección, ya que es necesario establecer propiedades de los objetos del tipo, y mediante ellas comprobar que los objetos que se manejan pertenecen realmente al tipo. Estas propiedades se formalizan en los llamados *invariantes de la representación* de la forma que se explica seguidamente. El sentido de esta segunda parte de la verificación es el de una comprobación de tipos exhaustiva: al escoger una representación, por ejemplo, las listas en el caso del tipo abstracto `tConj`, se activa la comprobación de tipos implícita de Pascal, es decir, el compilador se asegura de que todo objeto del tipo `tConj` es una lista del tipo `^tNodoLista`.

Pero esta comprobación de tipos implícita no es suficiente, por el hecho de que no toda lista de enteros es una representación legal de un conjunto, concretamente porque puede tener elementos repetidos. Por tanto, es necesaria una verificación más profunda que debe ser realizada por el programador-diseñador. Esta verificación se llevará a cabo formalizando las propiedades pertinentes de los objetos en los citados invariantes de representación.

Por ejemplo, si se decide representar los conjuntos de enteros mediante listas sin repetición, este invariante se podría formalizar como sigue:

$$\{\text{Inv. } \forall i, j \in I,\ si\ i \neq j,\ entonces\ l_i \neq l_j\}$$

donde I es el conjunto de "índices" correspondientes a los elementos de la lista, y l_i es el elemento que ocupa la posición i-ésima en la lista. De esta forma se expresa que no deben existir dos elementos iguales en la lista.

No hay normas generales para establecer el invariante de representación, ya que depende del tipo abstracto de datos y de la representación escogida. Incluso, en alguna ocasión, el invariante puede ser trivial, como ocurre en la representación de los conjuntos de enteros mediante árboles binarios de búsqueda: en este caso, como en la implementación de estos árboles no se repiten las entradas (véase el apartado 17.4.2), se tiene asegurado que no habrá elementos duplicados, con lo cual el invariante de representación se formaliza simplemente como:

$$\{\text{Inv. } cierto\}$$

Una vez conocida la forma del invariante de representación, es necesario asegurarse de que lo verifica la representación particular de todo objeto del tipo abstracto de datos. Esto es sencillo, partiendo del hecho de que todo objeto del tipo es obtenido a partir de las operaciones constructoras o mediante la aplicación de las operaciones de selección o de las propias del tipo. Por ello, basta con comprobar (de forma inductiva) que todo objeto generado por estas operaciones verifica el invariante, suponiendo que los eventuales argumentos del tipo también lo cumplen. Así, volviendo a la representación mediante listas, la operación `CrearConj` lo cumple trivialmente, ya que genera el conjunto vacío representado, valga la redundancia, por la lista vacía, y se verifica que

$$\forall i, j \in I,\ si\ i \neq j,\ entonces\ l_i \neq l_j$$

puesto que, en este caso, $I = \varnothing$.

En cuanto a la operación `AnnadirElemConj`, dado que el argumento recibido `conj` verifica el invariante, también lo verificará al completarse la ejecución, ya que en ella se comprueba explícitamente que `elem` no pertenece a `conj`.

El lector puede comprobar como ejercicio que el resto de las operaciones implementadas en el ejemplo también preservan el invariante.

El problema de la corrección en tipos abstractos de datos puede tratarse más formal y profundamente, pero ello escapa a las pretensiones de este libro. En las referencias bibliográficas se citan textos que profundizan en este tema.

Finalmente, se debe destacar que la modularidad introducida por los tipos abstractos de datos supone una gran ayuda en el estudio de la corrección de

programas grandes. Para éstos, dicho estudio resulta tan costoso que se ve muy reducido o incluso abandonado en la mayoría de los casos. Dada esta situación, es muy importante disponer de una biblioteca de tipos abstractos de datos correctos cuya verificación se ha realizado independientemente.

19.4 Resumen

En este apartado se enumeran algunas ideas que ayudan a fijar los conceptos expuestos en este capítulo, así como otros aspectos relacionados:

- Los tipos abstractos de datos permiten al programador concentrarse en las características o propiedades deseadas para dichos tipos (o en qué servicio deben prestar al programador), olvidándose, temporalmente, de cómo están implementados (o cómo se van a implementar). Por consiguiente, los tipos abstractos de datos pueden considerarse como cajas negras: el programador-usuario sólo ve su comportamiento y no sabe (ni le interesa saber) qué contienen.

- La definición del tipo abstracto de datos debe expresarse como una especificación no ambigua, que servirá como punto de partida para la derivación correcta de la implementación o para enfrentarla *a posteriori* a la implementación en un proceso de verificación formal.

- Para la aplicación de la abstracción de datos, es imprescindible que el lenguaje de programación escogido posibilite la implementación separada para poder llevar a cabo la ocultación de información y la "encapsulación", esenciales en esta técnica.

- La abstracción de datos optimiza los niveles de independencia (en la línea de lo comentado en los capítulos 8 y 9) del código de la unidad del tipo abstracto de datos y de los programas de aplicación. Como consecuencia de ello, aumenta la facilidad de mantenimiento de los programas, al aislar en la unidad de implementación del tipo abstracto los cambios, correcciones y modificaciones relativos a la representación del tipo abstracto, evitando que se propaguen (innecesariamente) a los programas de aplicación.

- La abstracción de datos se puede considerar como precursora de la técnica de orientación a objetos (véase el apartado 5.1.3 del tomo I), en la que también se encapsulan datos y operaciones (con la diferencia de que se añaden los mecanismos de herencia y polimorfismo, entre otros).[8]

[8] A partir de la versión 5.5 de Turbo Pascal se permite una aplicación limitada de esta técnica, pero su explicación excede los propósitos de este libro.

19.5 Ejercicios

1. Construya una unidad de Turbo Pascal para los siguientes tipos abstractos de datos. El tipo abstracto debe incluir operaciones de construcción, modificación y consulta, así como operaciones propias del tipo si se considera conveniente:

   (a) Listas (de enteros). Consúltese el apartado 17.1.

   (b) Listas ordenadas (de enteros). Consúltese el apartado 17.1.

   (c) Pilas (de enteros). Consúltese el apartado 17.2.

   (d) Colas (de enteros). Consúltese el apartado 17.3.

   (e) Árboles binarios (de enteros). Consúltese el apartado 17.4.

   (f) Árboles binarios de búsqueda, conteniendo en sus nodos cadenas de caracteres. Consúltese el apartado 17.4.2.

2. Desarrolle una implementación alternativa para el tipo abstracto `tConj`, basando la representación en:

   (a) Un vector de booleanos. Obviamente, en este caso se deben limitar los elementos posibles (supóngase por ejemplo, que los elementos están comprendidos entre 1 y 1000, ambos inclusive).

   (b) Árboles binarios de búsqueda (véanse los apartados 19.2.3 y 17.4.2).

   Compare el comportamiento que tienen las operaciones en ambos tipos abstractos de datos.

3. Implemente las operaciones `Union, Interseccion` y `Diferencia` (que calculan, obviamente, la unión, intersección y diferencia de dos conjuntos) como operaciones propias del tipo abstracto de datos `tConj`, representado mediante árboles binarios de búsqueda.

4. Programe un algoritmo que ordene una secuencia utilizando dos pilas. Para ello debe mover elementos de una pila a otra asegurándose de que los items menores llegan a la cima de una de las pilas y los mayores a la cima de la otra. Utilice el tipo abstracto pila desarrollado en el ejercicio 1c.

5. Escriba una función para sumar dos polinomios utilizando alguno de los tipos abstractos de datos propuestos en el ejercicio 1.

6. Escriba una unidad para la aritmética entera (sin límite en sus valores máximo y mínimo), utilizando alguno de los tipos abstractos de datos propuestos en el ejercicio 1.

7. Una *cola doble* es una estructura de datos consistente en una lista de elementos sobre la cual son posibles las siguientes operaciones:

   - *Meter(x,d)*: inserta el elemento x en el extremo frontal de la cola doble d.
   - *Sacar(x,d)*: elimina y devuelve el elemento que está al frente de la cola doble *d*.

- *Inyectar(x,d)*: inserta el elemento x en el extremo posterior de la cola doble d.
- *Expulsar(x,d)*: elimina y devuelve el elemento que está en el extremo posterior de la cola doble d.

Programar una unidad de Turbo Pascal para el tipo abstracto de datos *cola doble*.

19.6 Referencias bibliográficas

En este capítulo se introducen los tipos abstractos de datos, eludiendo las profundidades formales que hay tras ellos y evitando también dar catálogos exhaustivos de los tipos abstractos de datos más usuales. En [Mar86, Har89] se estudian los aspectos formales de los tipos abstractos de datos basados en especificaciones algebraicas. Además, ambos textos pueden considerarse como catálogos de los tipos abstractos de datos de más amplia utilización, incluyendo la especificación, implementaciones y aplicaciones comunes de cada uno de ellos. En [Pn93] se puede encontrar un estudio actual, riguroso y completo de los tipos abstractos de datos en nuestro idioma, junto con especificaciones de los más usuales. Igualmente, [HS90] es una lectura obligada en cuanto se aborda el estudio de tipos abstractos de datos.

Con un enfoque más aplicado, en [LG86, CMM87] se da un tratamiento completo de los tipos abstractos de datos bien adaptado a Pascal, ofreciendo propuestas de implementación en este lenguaje.

Por supuesto, los tipos abstractos de datos pueden ser implementados en lenguajes de programación diferentes de Pascal. De hecho, Pascal estándar no cuenta con mecanismos apropiados para manejarlos, ya que el concepto de tipo abstracto de datos es posterior a la creación de este lenguaje. El propio N. Wirth incluyó este concepto en su Módula2 a través de los módulos, y las unidades de Turbo Pascal no son más que un remedo (incompleto, por cierto) de este mecanismo. Otros lenguajes con facilidades para el desarrollo de tipos abstractos de datos son Ada [Bar87], que dispone de paquetes, y C++ [Str84], haciendo uso de clases. Además, algunos lenguajes proporcionan al programador técnicas de compilación separada más adecuadas a los tipos abstractos y sus necesidades que las incompletas unidades de Turbo Pascal.

Finalmente, debemos indicar que el ejercicio 4 está tomado de [Mar86]. Los ejercicios 5, 6 y 7 están tomados de [Wei95].

Capítulo 20

Esquemas algorítmicos fundamentales

Cuando se estudian los problemas y algoritmos usualmente escogidos para mostrar los mecanismos de un lenguaje algorítmico (ya sea ejecutable o no), puede parecer que el desarrollo de algoritmos es un cajón de sastre en el que se encuentran algunas ideas de uso frecuente y cierta cantidad de soluciones *ad hoc*, basadas en trucos más o menos ingeniosos.

Sin embargo, la realidad no es así: muchos problemas se pueden resolver con algoritmos construidos en base a unos pocos modelos, con variantes de escasa importancia. En este capítulo se estudian algunos de esos esquemas algorítmicos fundamentales, su eficiencia y algunas de las técnicas más empleadas para mejorarla.

20.1 Algoritmos devoradores

La estrategia de estos algoritmos es básicamente iterativa, y consiste en una serie de etapas, en cada una de las cuales se consume una parte de los datos y se construye una parte de la solución, parando cuando se hayan consumido totalmente los datos. El nombre de este esquema es muy descriptivo: en cada fase (bocado) se consume una parte de los datos. Se intentará que la parte consumida sea lo mayor posible, bajo ciertas condiciones.

20.1.1 Descripción

Por ejemplo, la descomposición de un número n en primos puede describirse mediante un esquema devorador. Para facilitar la descripción, consideremos la descomposición de 600 expresada así:

$$600 = 2^3\ 3^1\ 5^2$$

En cada fase, se elimina un divisor de n cuantas veces sea posible: en la primera se elimina el 2, y se considera el correspondiente cociente, en la segunda el 3, etc. y se finaliza cuando el número no tiene divisores (excepto el 1).

El esquema general puede expresarse así:

```
procedure Resolver P (D: datos; var S: solucion);
begin
  Generar la parte inicial de la solución S
  (y las condiciones iniciales)
  while D sin procesar del todo do begin
    Extraer de D el máximo trozo posible T
    Procesar T (reduciéndose D)
    Incorporar el procesado de T a la solución S
  end {while}
end; {Resolver P}
```

La descomposición de un número en factores primos se puede implementar sencillamente siguiendo este esquema:

```
procedure Descomponer(n: integer);
  {PreC.: n > 1}
  {Efecto: muestra en la pantalla la descomposición de n
  en factores primos}
  var
    d: integer;
```

```
begin
  d:= 2;
  while n > 1 do begin
    Dividir n por d cuantas veces (k) se pueda
    Escribir 'd^k'
    d:= d + 1
  end {while}
end; {Descomponer}
```

20.1.2 Adecuación al problema

Otro ejemplo, quizá el más conocido de este esquema, es el de encontrar el "cambio de moneda" de manera óptima (en el sentido de usar el menor número de monedas posible) para una cantidad de dinero dada:[1] en cada fase, se considera una moneda de curso legal, de mayor a menor valor, y se cambia la mayor cantidad de dinero posible en monedas de ese valor.

Se advierte, sin embargo, que no siempre es apropiado un algoritmo devorador para resolver este problema. Por ejemplo, si el sistema de monedas fuera de

1, 7 y 9 pesetas

el cambio de 15 pesetas que ofrece este algoritmo consta de

7 monedas: 1 de 9 y 6 de 1

mientras que el cambio óptimo requiere tan sólo

3 monedas: 2 de 7 y 1 de 1

El algoritmo presentado para el cambio de moneda resultará correcto siempre que se considere un sistema monetario donde los valores de las monedas son cada uno múltiplo del anterior. Si no se da esta condición, es necesario recurrir a otros esquemas algorítmicos (véase el apartado 20.3.3).

La enseñanza que extraemos del ejemplo es la siguiente: generalmente el desarrollo de esta clase de algoritmos no presenta dificultades, pero es complicado asegurarse de que esta técnica es apropiada para el problema planteado. Por otra parte, hay que señalar que este esquema se aplica con frecuencia en la resolución de problemas a sabiendas de que la solución proporcionada no es la óptima, sino sólo relativamente buena. Esta elección se debe a la rapidez de la resolución, circunstancia que muchas veces hace que no merezca la pena buscar una solución mejor.

[1] Para simplificar, supondremos que se dispone de cuantas monedas se necesite de cada valor.

20.1.3 Otros problemas resueltos vorazmente

Existen gran cantidad de problemas cuya solución algorítmica responde a un esquema voraz. Entre ellos, los dos siguientes son ampliamente conocidos.

Problema de la mochila

Se desea llenar una mochila hasta un volumen máximo V, y para ello se dispone de n objetos, en cantidades limitadas $v_1, \ldots, v_n$ y cuyos valores por unidad de volumen son $p_1, \ldots, p_n$, respectivamente. Puede seleccionarse de cada objeto una cantidad cualquiera $c_i \in \mathbb{R}$ con tal de que $c_i \leq v_i$. El problema consiste en determinar las cantidades $c_1, \ldots, c_n$ que llenan la mochila maximizando el valor $\sum_{i=1}^{n} v_i p_i$ total.

Este problema puede resolverse fácilmente seleccionando, sucesivamente, el objeto de mayor valor por unidad de volumen que quede y en la máxima cantidad posible hasta agotar el mismo. Este paso se repetirá hasta completar la mochila o agotar todos los objetos. Por lo tanto, se trata claramente de un esquema voraz.

Árbol de expansión mínimo (algoritmo de Prim)

El problema del árbol de expansión mínimo[2] se encuentra por ejemplo en la siguiente situación: consideremos un mapa de carreteras, con dos tipos de componentes: las ciudades (*nodos*) y las carreteras que las unen. Cada tramo de carreteras (*arco*) está señalado con su longitud.[3] Se desea implantar un tendido eléctrico siguiendo los trazos de las carreteras de manera que conecte todas las ciudades y que la longitud total sea mínima.

Una forma de lograrlo consiste en

Empezar con el tramo de menor coste
repetir
  Seleccionar un nuevo tramo
hasta que esté completa una red que conecte todas las ciudades

donde cada nuevo tramo que se selecciona es el de menor longitud entre los no redundantes (es decir, que da acceso a una ciudad nueva).

Un razonamiento sencillo nos permite deducir que un árbol de expansión cualquiera (el mínimo en particular) para un mapa de n ciudades tiene $n - 1$ tramos. Por lo tanto, es posible simplificar la condición de terminación que controla el algoritmo anterior.

[2]También llamado árbol de recubrimiento (del inglés, *spanning tree*.)

[3]Este modelo de datos se llama *grafo ponderado*.

20.2 Divide y vencerás

La idea básica de este esquema consiste en lo siguiente:

1. Dado un problema P, con datos D, si los datos permiten una solución directa, se ofrece ésta;
2. En caso contrario, se siguen las siguientes fases:
   (a) Se dividen los datos D en varios conjuntos de datos más pequeños, D_i.
   (b) Se resuelven los problemas $P(D_i)$ parciales, sobre los conjuntos de datos D_i, recursivamente.
   (c) Se combinan las soluciones parciales, resultando así la solución final.

El esquema general de este algoritmo puede expresarse así:

```
procedure Resolver P (D: datos; var S: solucion);
begin
  if los datos D admiten un tratamiento directo then
    Resolver P(D) directamente
  else begin
       Repartir D en varios conjuntos de datos, D_1, ..., D_k
           (más cercanos al tratamiento directo).
       Resolver P(D_1), ..., P(D_k)
           (llamemos S_1, ..., S_k a las soluciones obtenidas).
       Combinar S_1, ..., S_k, generando la solución, S, de P(D).
  end
end; {Resolver P}
```

Como ejemplo, veamos que el problema de ordenar un vector admite dos soluciones siguiendo este esquema: los algoritmos *Merge Sort* y *Quick Sort* (véanse los apartados 15.2.5 y 15.2.4, respectivamente).

Tal como se presentó en el apartado antes citado, el primer nivel de diseño de *Merge Sort* puede expresarse así:

```
si v es de tamaño 1 entonces
  v ya está ordenado
si no
  Dividir v en dos subvectores A y B
fin {si}
Ordenar A y B usando Merge Sort
Mezclar las ordenaciones de A y B para generar el vector ordenado.
```

Esta organización permite distinguir claramente las acciones componentes de los esquemas divide y vencerás comparándolo con el esquema general. El siguiente algoritmo, *Quick Sort*, resuelve el mismo problema siguiendo también una estrategia divide y vencerás:

si `v` *es de tamaño 1 entonces*
  `v` *ya está ordenado*
si no
  Dividir `v` *en dos bloques* `A` *y* `B`
  con todos los elementos de `A` *menores que los de* `B`
fin {`si`}
Ordenar `A` *y* `B` *usando Quick Sort*
Devolver `v` *ya ordenado como concatenación*
de las ordenaciones de `A` *y de* `B`

Aunque ambos algoritmos siguen el mismo esquema, en el primero la mayor parte del trabajo se efectúa al combinar las subsoluciones, mientras que en el segundo la tarea principal es el reparto de los datos en subconjuntos. De hecho, lo normal es operar reestructurando el propio vector, de modo que no es preciso combinar las subsoluciones concatenando los vectores. Además, como se analizó en el apartado 18.3.2, el algoritmo *Merge Sort* resulta ser más eficiente en el peor caso, ya que su complejidad es del orden de $n \log n$ frente a la complejidad cuadrática de *Quick Sort*, también en el peor caso.

20.2.1 Equilibrado de los subproblemas

Para que el esquema algorítmico divide y vencerás sea eficiente es necesario que el tamaño de los subproblemas obtenidos sea similar. Por ejemplo, en el caso del algoritmo *Quick Sort*, y en relación con estos tamaños, se podrían distinguir dos versiones:

- La presentada anteriormente, cuya complejidad en el caso medio es del orden de $n \log n$.
- La degenerada, en la que uno de los subproblemas es la lista unitaria es de tamaño 1, y en la que se tiene una complejidad cuadrática. De hecho, esta versión de *Quick Sort* es equivalente al algoritmo de ordenación por inserción (véase el apartado 15.2.2).

No obstante, hay problemas en los que el esquema divide y vencerás no ahorra coste, ni siquiera equilibrando los subproblemas. Por ejemplo, el cálculo de $\prod_{i=a}^{b} i$. En efecto, su versión recursiva con los subproblemas equilibrados

$$\prod_{i=a}^{b} i = \begin{cases} 1 & \text{si } b < a \\ \prod_{i=a}^{m} i * \prod_{i=m+1}^{b} i, \text{ para } m = (a+b) \textbf{ div } 2, & \text{e. o. c.} \end{cases}$$

tiene un coste proporcional a $b - a$, al igual que su versión degenerada, donde uno de los subproblemas es el trivial:

$$\prod_{i=a}^{b} i = \begin{cases} 1 & \text{si } b < a \\ a * \prod_{i=a+1}^{b} i & \text{e. o. c.} \end{cases}$$

siendo por tanto preferible su versión iterativa conocida.

20.3 Programación dinámica

20.3.1 Problemas de programación dinámica

Consideremos un problema en el que se desea obtener el valor óptimo de una función, y para ello se ha de completar una secuencia de etapas $e_1, \ldots, e_n$. Supongamos que en la etapa e_i se puede escoger entre las opciones $o_1, \ldots, o_k$, cada una de las cuales divide el problema en dos subproblemas más sencillos e independientes:

$$e_1, \ldots, e_{i-1} \quad \text{y} \quad e_{i+1}, \ldots, e_n$$

Entonces, bastaría con resolver los k pares de subproblemas (uno por cada opción para e_i) y escoger el par que ofrece el mejor resultado (para la función por optimizar).

Supongamos que la solución óptima se obtiene si $e_i = o$:

$$e_1, \ldots, e_{i-1}, e_i = o, e_{i+1}, \ldots, e_n$$

Entonces, sus mitades anterior y posterior

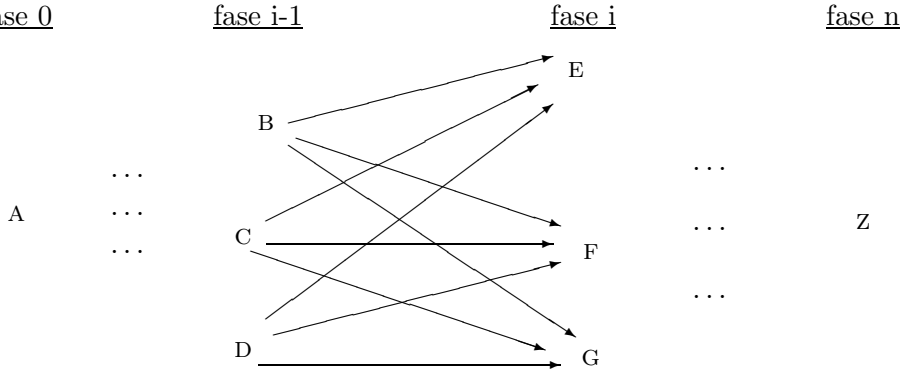
$$e_1, \ldots, e_{i-1}, o$$
$$o, e_{i+1}, \ldots, e_n$$

deben ser las soluciones óptimas a los subproblemas parciales planteados al fijar la etapa i-ésima. Esta condición se conoce como el *principio de optimalidad de Bellman*. Por ejemplo, si para ir desde el punto A hasta el punto C por el camino más corto se pasa por el punto B, el camino mínimo desde A hasta C consiste en la concatenación del camino mínimo desde A hasta B y el camino mínimo desde B hasta C.

Por tanto, la resolución de estos problemas consiste en definir una secuencia de etapas. Por otra parte, al fijar una etapa cualquiera se divide el problema en dos problemas más sencillos (como en los algoritmos divide y vencerás), que deben ser independientes entre sí. El principal inconveniente es que la elección de una etapa requiere en principio tantear varios pares de subproblemas, con lo que se dispara el coste de la resolución de estos problemas.

Ejemplo

Consideremos un grafo como el de la figura, organizado por fases,[4] y se plantea el problema de averiguar el recorrido más corto entre A y Z, conociendo las longitudes de los arcos:



El problema propuesto consiste en formar una secuencia de etapas, en cada una de las cuales se opta por un arco que conduce a uno de los nodos de la siguiente, accesible desde nuestro nodo actual, determinado por las etapas anteriores.

Si en la fase i-ésima se puede escoger entre los nodos E, F y G, el camino más corto entre A y Z es el más corto entre los tres siguientes

ir desde A *hasta* E*, e ir desde* E *hasta* Z
ir desde A *hasta* F*, e ir desde* F *hasta* Z
ir desde A *hasta* G*, e ir desde* G *hasta* Z

de manera que, una vez fijada una elección (por ejemplo E), las subsoluciones *ir desde A hasta E* e *ir desde E hasta Z* componen la solución global. Esto es, se verifica el principio de optimalidad de Bellman.

En resumen, el mejor trayecto entre los puntos P y Q (en fases no consecutivas f_{inic} y f_{fin}), se halla así:

Elegir una etapa intermedia i (por ejemplo, $(f_{inic} + f_{fin})$ **div** 2*)*
  (sean $\{o_1, \ldots, o_k\}$ *los puntos en la etapa i-ésima)*
Elegir $O \in \{o_1, \ldots, o_k\}$ *tal que el recorrido MejorTray(P,O)*
  junto con MejorTray(O,Q) sea mínimo

La recursión termina cuando los nodos a enlazar están en etapas consecutivas, existiendo un único tramo de P a Q (en cuyo caso se elige éste) o ninguno (en cuyo

[4]Este tipo de grafos se llaman *polietápicos*.

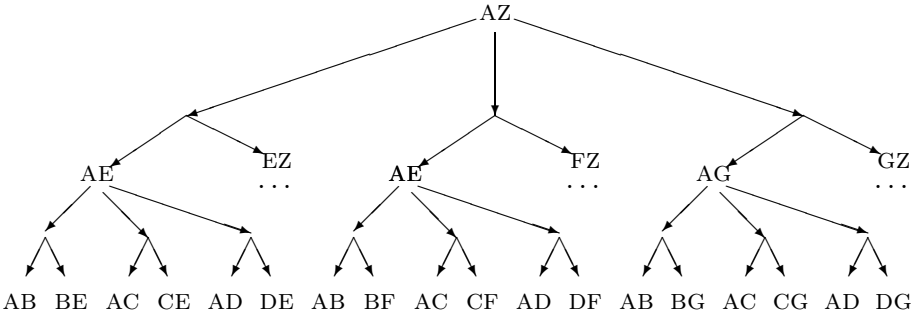
caso no hay trayecto posible entre los nodos a enlazar, lo que puede consignarse, por ejemplo, indicando una distancia infinita entre esos puntos).

Estas ideas pueden expresarse como sigue, desarrollando a la vez un poco más el algoritmo[5] descrito antes:

```
procedure MejorTray (P, Q: puntos; f_P, f_Q: nums.fase; var Tr: trayecto;
                     var Dist: distancia);
begin
  if consecutivos(P,Q) then
    Tr:=[P, Q]
    Dist:=longArco(P, Q), dato del problema
  else begin
    Sea f_med ← (f_P + f_Q) div 2,
      y sean {o_1, ..., o_k} los puntos de paso de la etapa i-ésima
    Se parte de distPQ ← ∞ y trPQ ← [ ]
    para todo O ∈ {o_1, ..., o_k} hacer begin
      MejorTray(P,O, f_P, f_med,TrPO, DistPO)
      MejorTray(O,Q, f_med, f_Q, TrOQ, DistOQ)
      if DistPQ > DistPO + DistOQ then begin
          DistPQ:= DistPO + DistOQ;
          TrPQ := TrPO concatenado con TrOQ
      end {if}
    end; {para todo}
    Tr:= TrPQ
    Dist:= DistPQ
  end {else}
end; {MejorTray}
```

20.3.2 Mejora de este esquema

El planteamiento anterior se caracteriza por su ineficiencia debida al gran número de llamadas recursivas. En el caso anterior, por ejemplo, el recorrido AZ puede descomponerse de múltiples formas, como se recoge en el siguiente árbol:



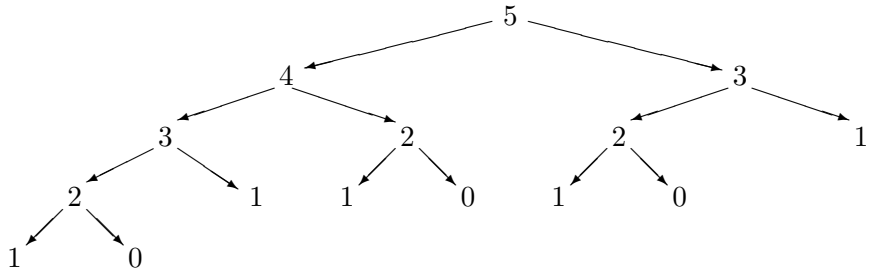
[5]Se advierte que este algoritmo es tremendamente ineficiente y, por tanto, nada recomendable; en el siguiente apartado se verán modos mejores de afrontar esta clase de problemas.

Sin embargo, se observa que un buen número de tramos se calcula repetidamente (AB, AC, AD, ...), por lo que se puede mejorar el planteamiento evitando los cálculos idénticos reiterados, concretamente mediante las técnicas de tabulación.

En este apartado estudiaremos en primer lugar en qué consisten esas técnicas, y después retomaremos el algoritmo anterior para ver cómo puede mejorarse su comportamiento mediante la tabulación.

Tabulación de subprogramas recursivos

En ocasiones, una función f con varias llamadas recursivas genera, por diferentes vías, llamadas repetidas. La función de Fibonacci (véase el apartado 10.3.1) es un ejemplo clásico:



Al aplicarse al argumento 5, la llamada `Fib(1)` se dispara cinco veces.

Una solución para evitar la evaluación repetida consiste en dotar a la función de memoria de modo que recuerde los valores para los que se ha calculado junto con los resultados producidos. Así, cada cálculo requerido de la función se consultará en la tabla, extrayendo el resultado correspondiente si ya se hubiera efectuado o registrándolo en ella si fuera nuevo.

El esquema es bien sencillo: basta con establecer una tabla (la memoria de la función) global e incluir en ella las consultas y actualizaciones mencionadas:

```
function f'(x: datos; var T: tablaGlobal): resultados;
begin
  if x no está en la tabla T then begin
    Hallar las llamadas recursivas f'(xi, T) y combinarlas, hallando
      el valor f'(x, T) requerido
    Incluir en la tabla T el argumento x y el resultado f'(x, T) obtenido
  end; {if}
  f':= T[x]
end; {f'}
```

Por ejemplo, la función de Fibonacci definida antes puede tabularse como sigue. En primer lugar, definimos la tabla:

```
type
  tDominio = 0..20;
  tTabla = array [tDominio] of record
               definido:  boolean;
               resultado:  integer
             end; {record}
var
  tablaFib:  tTabla;
```

cuyo estado inicial debe incluirse en el programa principal:

```
tablaFib[0].definido:= True;
tablaFib[0].resultado:= 1;
tablaFib[1].definido:= True;
tablaFib[1].resultado:= 1;
for i:= 2 to 20 do
  tablaFib[i].definido:= False
```

Entonces, la función `Fib` resulta

```
function Fib(n:  tDominio; var t:  tTabla):  integer;
begin
  if not t[n].definido then begin
    t[n].definido:= True;
    t[n].resultado:= Fib(n-1,t) + Fib(n-2,t)
  end; {if}
  Fib:= t[n].resultado
end; {Fib}
```

que se llama, por ejemplo, mediante `Fib(14, tablaFib)`.

Un requisito indispensable para que una función se pueda tabular correctamente es que esté libre de (producir o depender de) efectos laterales, de manera que el valor asociado a cada argumento sea único, independiente del punto del programa en que se requiera o del momento en que se invoque. Esta observación es necesaria, ya que en esta técnica se utilizan variables globales; como se explicó en el apartado 8.5.4, un uso incontrolado de estas variables puede acarrear la pérdida de la corrección de nuestros programas.

Tabulación de algoritmos de programación dinámica

Veamos ahora cómo las técnicas de tabulación descritas mejoran la eficiencia del algoritmo del grafo polietápico.

En efecto, se puede alterar el procedimiento descrito de modo que cada operación realizada se registre en una tabla y cada operación por realizar se consulte previamente en esa tabla por si ya se hubiera calculado. En el ejemplo anterior, se podría anotar junto a los puntos del propio mapa el mejor trayecto que conduce a ellos cada vez que se calcule. De este modo, no será necesario repetir esos cálculos cuando vuelvan a requerirse. Los cambios descritos son mínimos:

- Se crea una tabla global (donde registraremos los mejores trayectos y las correspondientes distancias), y se rellena inicialmente con los trayectos y distancias de los puntos consecutivos (que son los datos del problema).
- Entonces en vez de comprobar si un camino es directo, se hará lo siguiente:

  si consecutivos(P,Q) entonces
  el mejor trayecto es la secuencia [P,Q] y su longitud es longArco(P,Q)

  comprobaremos si está ya tabulado:

  si tabulado(P,Q) entonces
  extraer de la tabla el trayecto, trPQ, y su longitud, distPQ
  ...

- Por último, al finalizar un cálculo, se debe añadir a la tabla la acción siguiente:

  Registrar en la tabla el trayecto (TrPQ) y la distancia (distPQ) hallados

Esta solución es bastante satisfactoria. Sin embargo, este tipo de problemas permite, en general, establecer un orden entre los subproblemas requeridos y, por consiguiente, entre los cálculos necesarios. En el ejemplo anterior, los cálculos pueden hacerse por fases: una vez hallado (y anotado) el mejor trayecto que lleva a cada uno de los puntos de una fase i, es sencillo y rápido hallar (y anotar) los mejores caminos hasta los puntos de la fase siguiente, $i+1$. Más aún, como ya no se necesita la información correspondiente a la fase i-ésima, es posible prescindir de ella, con el consiguiente ahorro de memoria.

20.3.3 Formulación de problemas de programación dinámica

El problema del cambio de moneda (véase el apartado 20.1.2) se puede formular también de modo que los valores de las monedas no guardan relación alguna: se dispone de una colección ilimitada de monedas, de valores $v_1, \ldots, v_n$

enteros cualesquiera, y se trata ahora de dar el cambio óptimo para la cantidad C, entendiendo por óptimo el que requiere el menor número total de monedas.[6]

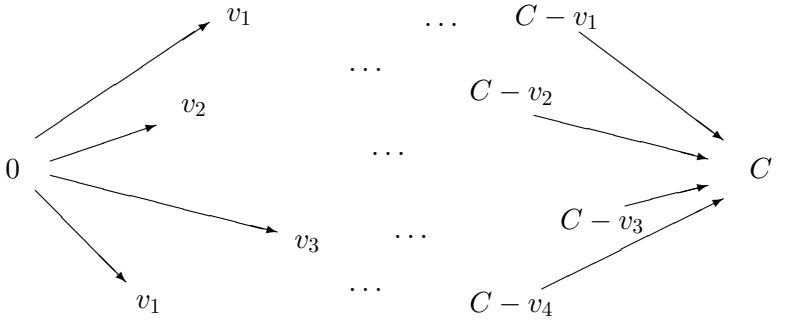
Una formulación siguiendo el esquema de programación dinámica es la siguiente: si hay alguna moneda de valor igual a la cantidad total, el cambio óptimo es precisamente con *una* moneda; de lo contrario, el primer paso consiste en escoger una moneda entre las de valor menor que la cantidad dada y esa elección debe minimizar el cambio de la cantidad restante:

```
function NumMon(C: integer):  integer;
begin
  if alguna de las monedas v_1, ..., v_n es igual a C then
    NumMon:= 1
  else
    NumMon:= 1 + mín_{v_i<C} (NumMon(C-v_i))
end; {NumMon}
```

La tabulación es sencilla: para obtener el cambio óptimo de una cantidad C, será necesario consultar (posiblemente varias veces) los cambios de cantidades menores. Por lo tanto, es fácil establecer un orden entre los subproblemas, tabulando los cambios correspondientes a las cantidades $1, 2, \ldots C$, ascendentemente.

Este ejemplo nos permite extraer dos consecuencias:

- En algunos problemas no se enuncian explícitamente las etapas que deben superarse desde el estado inicial a una solución; más aún, a veces el problema no consta de un número de fases conocido de antemano, sino que éstas deben ser "calculadas" por el algoritmo propuesto, como se muestra gráficamente en la siguiente figura:



[6]Para simplificar, supondremos que una de las monedas tiene el valor unidad, de manera que siempre es posible completar el cambio.

- El ejemplo presentado obedece a un planteamiento matemático con aplicaciones en campos muy diversos. Concretamente, si llamamos $k_1, \ldots, k_n$ al número de monedas de cada valor $v_1, \ldots, v_n$ respectivamente, el esquema anterior resuelve el siguiente problema de optimización:

$$mín \sum_{i=1}^{n} k_i \quad \text{sujeto a que} \quad \sum_{i=1}^{n} k_i v_i = C$$

que es muy similar al planteamiento general numérico de esta clase de problemas: hallar enteros no negativos $x_1, \ldots, x_n$ que minimicen la función $g(k_1, \ldots, k_n)$ definida así:

$$g(k_1, \ldots, k_n) = \sum_{i=1}^{n} f_i(k_i)$$

y de manera que se mantenga $\sum_{i=1}^{n} k_i v_i \leq C$.

Nota final

Es importante subrayar que un problema planteado no siempre admite una descomposición en subproblemas independientes: en otras palabras, antes de expresar una solución basada en tal descomposición, debe comprobarse que se verifica el principio de optimalidad.

En general, la resolución de esta clase de problemas resulta inviable sin la tabulación. Además, frecuentemente resulta sencillo fijar un orden en los cómputos con lo que, a veces, es posible diseñar un algoritmo iterativo de resolución.

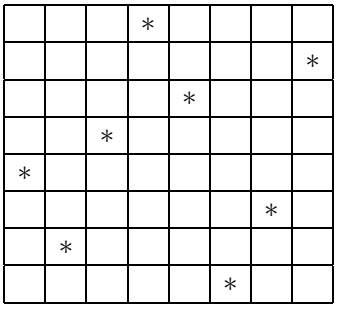
20.4 Vuelta atrás

Consideremos el problema de completar un rompecabezas. En un momento dado, se han colocado unas cuantas piezas, y se tantea la colocación de una nueva pieza. Por lo general, será posible continuar de diversos modos, y cada uno de ellos podrá ofrecer a su vez diversas posibilidades, multiplicándose así las posibilidades de tanteo. La búsqueda de soluciones es comparable al recorrido de un árbol, por lo que se le llama *árbol de búsqueda* (véase el apartado 17.5) o también *espacio de búsqueda*.

Por otra parte, el tanteo de soluciones supone muchas veces abandonar una vía muerta cuando se descubre que no conduce a la solución, deshaciendo algunos movimientos y regresando por otras ramas del árbol de búsqueda a una posición anterior. De ahí viene la denominación de esta clase de algoritmos: *vuelta atrás* o búsqueda con *retroceso*.[7]

[7] En inglés, *backtrack* o *backtracking*.

Un ejemplo conocido de problema que se adapta bien a este esquema es el de situar ocho damas en un tablero de ajedrez de manera que ninguna esté amenazada por otra. La siguiente figura muestra una solución:



Como cada dama debe estar en una fila distinta, es posible representar una solución como un vector $(D_1, \ldots, D_8)$, donde cada componente D_i es un entero de $\{1, \ldots, 8\}$ que representa la columna en que se encuentra la dama i-ésima.

La figura 20.1 muestra un fragmento del árbol de búsqueda en una fase intermedia en que se está construyendo la solución mostrada. Se ha usado el símbolo $\emptyset$ para representar el fallo en un intento de ampliar la solución por una rama, y obedece a la imposibilidad de situar una dama en ciertas casillas por estar a tiro de las situadas con anterioridad.

Las variantes más conocidas a este tipo de problemas son las siguientes:

1. Generar *todas* las soluciones posibles.

   El esquema de búsqueda de *todas* las soluciones, desde la fase i-ésima, se puede resumir como sigue:

```
procedure P (i: numero de fase; S: solucParcial);
    {Efecto: genera todas las soluciones, desde la fase i-ésima, a partir
     de la solución parcial S}
begin
  para todo p_i ∈ {posibles pasos válidos en esta fase } hacer begin
    Añadir p_i a S (Sea S' la solución S, dando un paso más p_i)
    if p_i completa una solución then
      Registrar la solución S´
    else
      Resolver P(i+1, S')
  end {para todo}
end; {P}
```

   En particular, el procedimiento que genera todas las soluciones en el problema de las ocho damas es el siguiente:

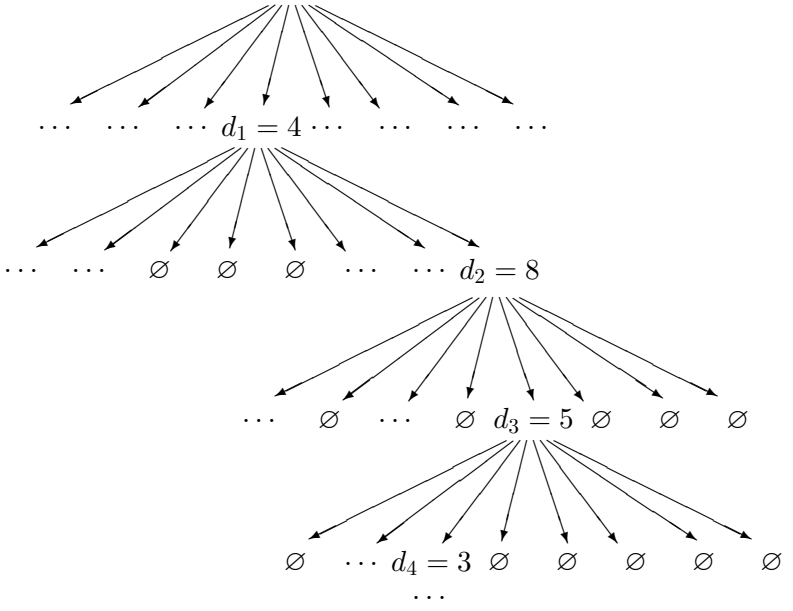


Figura 20.1.

```
type
  tDominio = 1..8;
...
procedure OchoDamas (i: numero de dama; S: solucParcial);
  var
    col:  tDominio;
begin
  for col:= 1 to 8 do
    if puede situarse la dama i-ésima en la columna col sin
       estar a tiro de las anteriores (1, ..., i-1) then begin
       Situar la dama i-ésima en la columna col
           (con lo que se tiene la situación S')
       if i = 8 then
                Registrar esta solución
       else
                Resolver OchoDamas (i+1, S')
    end {if}
end; {OchoDamas}
```

2. Tantear *cuántas* soluciones existen.

   El esquema es una variante del anterior, estableciendo un contador a cero antes de iniciar la búsqueda de soluciones y cambiando la acción de *registrar una solución* por incrementar el contador.

3. Generar *una* solución, si existe, o indicar lo contrario.

   En este caso, la búsqueda termina cuando se halla una solución o se agotan las vías posibles.

   Un modo fácil de conseguirlo es modificar el primer esquema añadiendo un parámetro (booleano) para indicar cuándo se ha encontrado una solución y parando la búsqueda en caso de éxito. En el caso del problema de las ocho damas quedaría así:

```
procedure OchoDamas(i: numero de dama; S: solucParcial;
                    var halladaSol:  boolean);
  var
    col:  tDominio;
begin
  halladaSol:= False;
  col:= 0;
  repeat
    col:= col + 1;
    if puede situarse la dama i-ésima en la columna col sin
       estar a tiro de las anteriores (1, ..., i-1) then begin
```

```
        Situar la dama i-ésima en la columna col
            (con lo que se tiene la situación S')
        if i = 8 then begin
            Registrar esta solución;
            halladaSol:= True
        end {then}
        else
          Resolver OchoDamas (i+1, S', halladaSol)
      end {then}
    until halladaSol or (col = 8)
  end; {OchoDamas}
```

En resumen, la técnica de vuelta atrás ofrece un método para resolver problemas tratando de completar una o varias soluciones por etapas $e_1, e_2, ..., e_n$ donde cada e_i debe satisfacer determinadas condiciones. En cada paso se trata de extender una solución parcial de todos los modos posibles, y si ninguno resulta satisfactorio se produce la vuelta atrás hasta el último punto en que aún quedaban alternativas sin explorar. El proceso de construcción de una solución es semejante al recorrido de un árbol: cada decisión ramifica el árbol y conduce a posiciones más definidas de la solución o a una situación de fallo.

20.4.1 Mejora del esquema de vuelta atrás

El árbol de búsqueda completo asociado a los esquemas de vuelta atrás crece frecuentemente de forma exponencial conforme aumenta su altura. Por ello, cobra gran importancia podar algunas de sus ramas siempre que sea posible. Los métodos más conocidos para lograrlo son los siguientes:

Exclusión previa

Frecuentemente, un análisis detallado del problema permite observar que ciertas subsoluciones resultarán infructuosas más o menos pronto. Estos casos permiten organizar el algoritmo para que ignore la búsqueda en tales situaciones.

El problema de las ocho damas es un claro ejemplo de ello: es obvio que cada dama deberá estar en una fila distinta, lo que facilita la organización de las fases y limita la búsqueda de soluciones.

Fusión de ramas

Cuando la búsqueda a través de distintas ramas lleve a resultados equivalentes, bastará con limitar la búsqueda a una de esas ramas.

En el problema de las ocho damas, por ejemplo, se puede limitar el recorrido de la primera dama a los cuatro primeros escaques, ya que las soluciones correspondientes a los otros cuatro son equivalentes y deducibles de los primeros por simetría.

Reordenación de la búsqueda

Cuando lo que se busca no son todas las soluciones, sino sólo una, es interesante reorganizar las ramas, situando en primer lugar las que llevan a un subárbol de menor tamaño o aquéllas que ofrecen mayores espectativas de éxito.

Ramificación y poda

Con frecuencia, el método de búsqueda con retroceso resulta impracticable debido al elevado número de combinaciones de prueba que aparecen. Cuando lo que se busca es precisamente *una* solución *óptima*, es posible reducir el árbol de búsqueda: suprimiendo aquellas fases que, con certeza, avanzan hacia soluciones no óptimas, por lo que se puede abandonar la búsqueda por esas vías (ésta es la idea que refleja el nombre[8] de esta técnica). Como consecuencia de esta reducción del número de soluciones por inspeccionar, se puede producir una mejora sustancial en la eficiencia de los algoritmos de búsqueda con retroceso convirtiéndolos en viables.

Por ejemplo, consideremos nuevamente el problema de las ocho damas, pero con la siguiente modificación: los escaques del tablero están marcados con enteros positivos, y se trata de hallar la posición de las damas que, sin amenazarse entre sí, cubre casillas cuya suma sea mínima. Ahora, si suponemos que se ha encontrado ya una solución (con un valor suma S), en el proceso de búsqueda de las posiciones siguientes se puede "podar" en cuanto una fase intermedia rebase la cantidad S, ya que se tiene la seguridad de que la solución obtenida al extender esa solución no mejorará en ningún caso la que ya tenemos.

Este ejemplo nos permite resaltar la conveniencia de reordenar la búsqueda: es obvio que, cuanto menor sea el valor ofrecido por una solución provisional, mayores posibilidades se tendrá de efectuar podas en las primeras fases. En nuestro ejemplo, las posibilidades de reorganizar la búsqueda son dos: establecer un orden entre las fases y dentro de cada fase.

[8] En inglés, *branch and bound*.

20.5 Anexo: algoritmos probabilistas

Por desgracia, los esquemas anteriores no siempre proporcionan soluciones a cualquier problema planteado; además sucede a veces que, aun existiendo una solución algorítmica a un problema, el tiempo requerido para encontrar una solución es tal que los algoritmos resultan inservibles. Una posibilidad consiste en conformarse con una solución aproximada, o bien con una solución que resulte válida sólo casi siempre. Ambas posibilidades se basan en la aplicación de métodos estadísticos.

En realidad, este planteamiento no constituye un esquema algorítmico, sino más bien una trampa consistente en alterar el enunciado de un problema intratable computacionalmente para ofrecer una respuesta aceptable en un tiempo razonable.

En este anexo incluimos sólo un par de ejemplos de los dos criterios más frecuentemente adoptados en esta clase de algoritmos:

- El primero consiste en buscar una *solución aproximada*, lo que es posible en bastantes problemas numéricos, sabiendo además el intervalo de confianza de la solución encontrada. La precisión de la solución depende frecuentemente del tiempo que se invierta en el proceso de cálculo, por lo que puede lograrse la precisión que se desee a costa de una mayor inversión del tiempo de proceso.

- El segundo criterio consiste en buscar una respuesta que constituya una solución del problema con cierta probabilidad. La justificación es que cuando se trata de tomar una decisión entre, por ejemplo, dos opciones posibles, no cabe pensar en soluciones aproximadas, aunque sí en *soluciones probablemente acertadas*. También aquí es posible invertir una cantidad de tiempo mayor para aumentar, esta vez, la probabilidad de acierto.

Aunque existen clasificaciones sistemáticas más detalladas de esta clase de algoritmos, en este apartado sólo pretendemos dar una idea del principio en que se basan, y ello mediante dos ejemplos ampliamente conocidos.

20.5.1 Búsqueda de una solución aproximada

Deseamos hallar el área de un círculo de radio 1 (naturalmente, sin usar la fórmula). Un modo poco convencional de lograrlo consiste en efectuar n lanzamientos al azar al cuadrado $[-1,1]\times[-1,1]$ y concluir con que la proporción de ellos que caiga dentro del círculo será proporcional a este área. En otras palabras, el procedimiento consiste en generar n puntos (p_x, p_y) aleatoriamente, extrayendo ambas coordenadas del intervalo $[-1,1]$ uniformemente:

```
function NumAciertos(numPuntos: integer): integer;
  {Dev. el número de lanzamientos que caen dentro del círculo}
  var
    i, total: integer;
begin
  total:= 0;
  for i:= 1 to numPuntos do begin
    GenerarPunto(p_x, p_y); {del intervalo [-1,1]^2, uniformemente}
    if (p_x, p_y) ∈ círculo de radio 1 then
      total:= total + 1
  end; {for}
  NumAciertos:= total
end; {NumAciertos}
```

Según la ley de los grandes números, si `númPuntos` es muy grande la proporción de aciertos tenderá a la proporción del área del círculo:

$$\frac{numAciertos}{numEnsayos} \rightarrow \frac{\pi}{4}$$

De hecho, este algoritmo suele presentarse como un modo de estimar π:

$$\pi \simeq 4 * \frac{numAciertos}{numEnsayos}$$

20.5.2 Búsqueda de una solución probablemente correcta

Se desea averiguar si un entero n es o no primo. Supuesto n impar,[9] el método convencional consiste en tantear los divisores $3, 5, \ldots, \texttt{Trunc}(\sqrt{n})$. Los $\frac{\texttt{Trunc}(\sqrt{n})}{2} - 1$ tanteos que requiere el peor caso, hacen este método inviable cuando se trata de un n grande.

Una alternativa es la siguiente: supongamos que se tiene un test[10]

```
function Test(n: integer): boolean;
```

que funciona aleatoriamente como sigue: aplicado a un número primo, resulta ser siempre `True`; por el contrario, aplicado a un número compuesto lo descubre (resultando ser `False`) con probabilidad p (por ejemplo, $\frac{1}{2}$). Por lo tanto, existe peligro de error sólo cuando su resultado es `True`, y ese resultado es erróneo con una probabilidad $q = 1 - p$.

[9] De lo contrario el problema ya está resuelto.

[10] La naturaleza del test no importa en este momento. Bástenos con saber que, efectivamente, existen pruebas eficientes de esta clase.

Entonces, la aplicación repetida de esa función

```
function RepTest(n, k: integer): boolean;
  var
    i: integer;
    pr: boolean;
begin
  pr:= False;
  i:= 0;
  repeat
    i:= i + 1;
    pr:= Test(n)
  until pr or (i = k);
  RepTest:= pr
end; {RepTest}
```

se comporta así: aplicada a un primo, resulta ser siempre `True`, y aplicada a un número compuesto lo descubre con una probabilidad $1 - q^k$. La probabilidad de error cuando el resultado es `True`, q^k, puede disminuirse tanto como se quiera a costa de aumentar el número de aplicaciones de la función `test`.

20.6 Ejercicios

1. Desarrollar completamente el programa para descomponer un entero positivo en sus factores primos descrito en el apartado 20.1.1, escribiendo finalmente la solución en la siguiente forma:

600	2
300	2
150	2
75	3
25	5
5	5
1	

2. Escribir el algoritmo de cambio de moneda descrito en apartado 20.1.2 ajustándose al esquema devorador general.

3. Desarrolle completamente un procedimiento devorador para el problema de la mochila descrito en el apartado 20.1.3.

4. Desarrolle completamente un procedimiento devorador para el algoritmo de Prim del árbol de recubrimiento mínimo. Téngase en cuenta la observación hecha al final del apartado 20.1.3 que nos permite simplificar la condición de terminación del bucle.

5. Se tienen dos listas A y B ordenadas. Escribir un algoritmo devorador que las mezcle produciendo una lista C ordenada.

6. Supongamos que el vector V consta de dos trozos $V_{1,\ldots,k}$ y $V_{k+1,\ldots,n}$ ordenados. Escribir un algoritmo devorador que los mezcle, produciendo un vector $W_{1,\ldots,n}$ ordenado.

7. Se desea hallar el máximo valor de un vector V de n enteros. Siguiendo una estrategia divide y vencerás,[11] una solución consiste en lo siguiente: si el vector tiene una sola componente, ésa es la solución; de lo contrario, se procede como sigue:

   (a) se divide el vector $V_{1,\ldots,n}$ en dos trozos: $A = V_{1,\ldots,k}$ y $B = V_{k+1,\ldots,n}$

   (b) se hallan los máximos para A y B respectivamente

   (c) se combinan las soluciones parciales M_A y M_B obtenidas, resultando que la solución final es el máximo de entre M_A y M_B.

   Desarrolle completamente el algoritmo.

8. (a) Desarrolle otro algoritmo que halle el máximo valor de un vector, esta vez siguiendo un esquema recursivo simple en el que se compare el primer elemento del vector con el valor máximo del resto del vector.

   (b) Compare la complejidad de este algoritmo con el del ejercicio anterior.

9. Sea A una matriz cuadrada de $n \times n$. Si n es par, A puede descomponerse en cuatro submatrices cuadradas, así:

$$A = \left(\begin{array}{c|c} A' & B' \\ \hline C' & D' \end{array}\right)$$

   y su determinante puede hallarse así:

$$| A | = | A' | * | D' | - | B' | * | C' | .$$

   Escriba un algoritmo divide y vencerás para hallar el determinante de una matriz de dimensión n, siendo n una potencia de 2.

10. (a) Desarrolle un programa que halle `Fib(15)` según la definición recursiva usual de la función de Fibonacci, y modifíquelo para que cuente el número de llamadas `Fib(0)`, `Fib(1)`, `...` que se efectúan.

    (b) Implemente una variante de la función del apartado (a), en la que se le dote de memoria.

11. Defina en Pascal la función *ofuscación* de Dïjkstra,

$$\begin{array}{ll} \text{Ofusc}(0) = & 0 \\ \text{Ofusc}(1) = & 1 \\ \text{Ofusc}(n) = & \left\{ \begin{array}{ll} \text{Ofusc}(\frac{n}{2}), & \text{si } n \text{ es par} \\ \text{Ofusc}(n+1) + \text{Ofusc}(n-1), & \text{si } n \text{ es impar} \end{array} \right\}, \forall n \geq 2 \end{array}$$

   dotándola de memoria, y averigüe en qué medida ha mejorado su eficiencia, siguiendo los pasos del ejercicio anterior.

[11] Aunque el algoritmo descrito resuelve correctamente el problema planteado, debe advertirse que ésta no es la manera más apropiada para afrontar este problema (véase el apartado 20.2.1).

12. Redefina la función que halla recursivamente los números combinatorios $\binom{m}{n}$, para $m \in \{0, \ldots, 10\}, n \in \{0, \ldots, m\}$, dotándola de memoria.

13. Desarrolle completamente el algoritmo que halla el camino mínimo en un grafo polietápico incorporando la tabulación descrita en el apartado 20.3.2

14. (a) Desarrolle completamente el problema del cambio de moneda descrito en el apartado 20.3.3 para el juego de monedas de 1, 8 y 9 pesetas. Dar dos versiones del mismo: una sin tabular y otra mediante tabulación.

    (b) Inserte los contadores necesarios para tantear el número de llamadas recursivas requeridas para descomponer 70 ptas. de manera óptima en las dos versiones anteriores.

15. Desarrolle un algoritmo que dé la lista de todas las descomposiciones posibles de `N` (número natural que es un dato de entrada) en sumas de doses, treses y cincos.

16. Supongamos que las soluciones parciales al problema de las ocho damas se concretan en dos variables: un indicador del número de damas situadas correctamente (con un valor entre 0 y 8), y un **array** `[1..8]` **of** `1..8` cuyas primeras casillas registran las posiciones de las damas ya situadas en una subsolución.

    (a) Escriba una función que indique si dos damas están a tiro.

    (b) Escriba una función que, conocida una subsolución y una posición (de 1 a 8) candidata para situar la dama siguiente, indique si es ello posible sin amenazar las anteriores.

    (c) Desarrolle completamente el algoritmo descrito en el apartado 1, generalizado al problema de las n *damas*, con tres variantes:

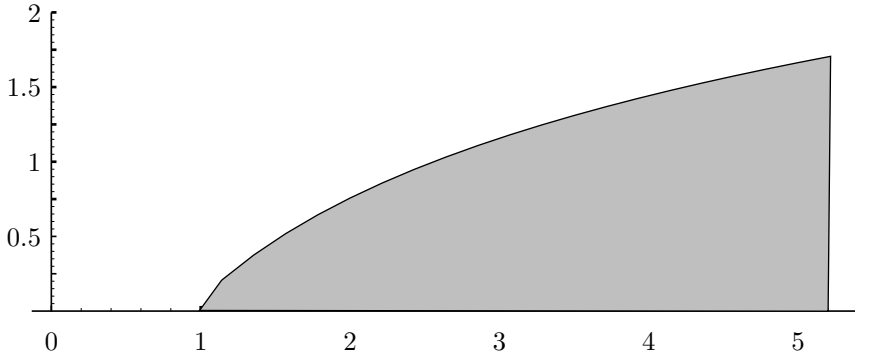
        i. La generación de *todas* las posiciones válidas.

        ii. Averiguar *cuántas* soluciones válidas hay.

        iii. Buscar *una* posición válida, si existe, o un mensaje de advertencia en caso contrario.

    (d) Aplique las técnicas de poda expuestas en el apartado 20.4.1 para situar ocho damas, libres de amenazas, en un tablero de ocho por ocho, ocupando casillas con una suma máxima, sabiendo que la casilla de la fila i, columna j, vale $10^{8-i} + j$ puntos.

17. Utilizando las técnicas explicadas en el apartado 20.5.1, hallar aproximadamente el área limitada por la función $f(x) = \log(x)$, el eje de abscisas y las rectas $x = 1$ y $x = 5$. Compárese el resultado obtenido con el valor exacto de $\int_1^5 \log(x)dx$.



18. **Experimento de Buffon**

    Supongamos un entarimado de parquet formado por tablillas paralelas de 10 cm de ancho.

    (a) Hallar, mediante simulación, la probabilidad de que, al lanzar una aguja de 5 cm de largo, caiga sobre una línea entre dos baldosas.

    (b) Sabiendo que esa probabilidad es $\frac{1}{\pi}$, estimar π de ese modo.

20.7 Referencias bibliográficas

Se introduce sucintamente en este capítulo un tema que requiere, para ser tratado cabalmente, un libro completo, por lo que los interesados en el tema necesitarán ampliar esta introducción. Afortunadamente, existe una enorme cantidad de referencias de calidad dedicada al estudio del diseño de los esquemas algorítmicos fundamentales, por lo que nos ceñimos a las fuentes en castellano a las que más debemos: los apuntes manuscritos de D. de Frutos [Fru84], los clásicos [AHU88], [HS90] y [BB97] y el más reciente [GGSV93].

Los ejemplos escogidos aquí para explicar cada uno de los esquemas algorítmicos no son, ciertamente, originales: al contrario, su utilización es frecuentísima con este fin, y ello se debe a que requieren un número pequeño de ideas para explicar, casi por sí mismos, los importantes conceptos que se han expuesto en este capítulo . En [Bie93] se abunda en esta idea, llegando a proponer el uso de estos ejemplos convertidos en pequeños rompecabezas cuyo manejo de forma natural consiste en cada uno de los esquemas algorítmicos fundamentales.

Los esquemas devorador y de programación dinámica se presentan frecuentemente usando como ejemplo el problema *de la mochila 0-1* (o mochila entera), alternativo al del cambio de moneda. Ambos pueden considerarse formulaciones del mismo argumento con distinta ambientación, admiten interesantes variaciones y han suscitado gran cantidad de observaciones. Además de las referencias citadas con carácter general, en [Wri75] pueden

leerse diferentes soluciones a los mismos; en [CK76] se estudian además las condiciones generales en que estos problemas admiten soluciones voraces.

Desde que apareció el problema de las n reinas (hace más de cien años), se han estudiado muchas de sus propiedades para tableros de distintos tamaños. En [BR75] puede encontrarse una visión panorámica del mismo y de otro problema clásico (la teselación con *pentominós*), ejemplificando con ambos el uso y mejora de las técnicas de vuelta atrás.

La referencia [Dew85a] es una bonita introducción a los algoritmos probabilistas. [BB90] ofrece una visión más detallada y profunda. El test para descubrir números compuestos descrito en 20.5.2 se encuentra en casi cualquier referencia sobre algoritmos probabilistas. Entre la bibliografía seleccionada, [BB97] y [Wil89] ofrecen claras explicaciones de los mismos.

Apéndices

Apéndice A

Aspectos complementarios de la programación

En este capítulo se estudian algunos aspectos que, aunque no se pueden considerar esenciales para una introducción a la programación en Pascal, sí resultan interesantes como complemento a los temas ya estudiados.

En primer lugar se analiza la posibilidad de definir en Pascal subprogramas con parámetros que son, a su vez, subprogramas. Con esto se pueden conseguir programas con un alto nivel de flexibilidad. Finalmente, se estudia la utilización de variables (seudo)aleatorias, que resultan especialmente útiles para desarrollar programas en los que interviene el azar.

A.1 Subprogramas como parámetros

La utilización de procedimientos y funciones como parámetros eleva a un nivel superior la potencia y versatilidad, ya conocida, de los propios subprogramas. Supongamos, por ejemplo, que disponemos de dos funciones $f, g : \mathbb{R} \to \mathbb{R}$, y queremos generar a partir de ellas una tercera que convierta cada $x \in \mathbb{R}$ en el máximo entre $f(x)$ y $g(x)$, como se muestra gráficamente en la figura A.1.

La dificultad del problema radica en que la función que deseamos definir no halla el máximo de dos números reales dados sino que, dadas dos funciones cualesquiera f y g y un número real x, halla $f(x)$ y $g(x)$ (aplicando las funciones al real) y obtiene el máximo de esos valores:

$$\texttt{MaxFuncs}(f, g, x) = \begin{cases} f(x) & \text{si } f(x) < g(x) \\ \\ g(x) & \text{en otro caso} \end{cases}$$

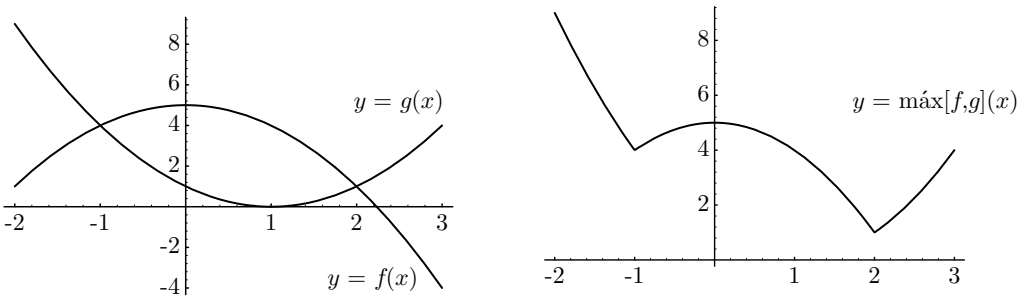


Figura A.1.

O sea, la función `MaxFuncs` responde al siguiente esquema:

$$\begin{array}{lcccccccc} \texttt{MaxFuncs} & : & (\mathbb{R} \to \mathbb{R}) & \times & (\mathbb{R} \to \mathbb{R}) & \times & \mathbb{R} & \to & \mathbb{R} \\ \texttt{MaxFuncs} & (& f & , & g & , & x &) = & \ldots \end{array}$$

La novedad consiste en que, si se quiere implementar un programa que resuelva el problema, dos de los parámetros serían subprogramas (concretamente, funciones). Esto está permitido en Pascal y para ello se incluye en la lista de parámetros formales el encabezamiento completo del subprograma parámetro,

```
function MaxFuncs (function F (x: real): real;
                   function G (x: real): real; x: real): real;
  {Dev.  el máximo de F(x) y G(x)}
begin
  if F(x) > G(x) then
    MaxFuncs:= F(x)
  else
    MaxFuncs:= G(x)
end; {MaxFuncs}
```

y en la llamada, como parámetro real, se coloca el identificador del subprograma que actúa como argumento:

```
y:= MaxFuncs (Sin, Cos, Pi/4)
```

En la ejecución de la llamada, al igual que ocurre con cualquier tipo de parámetros, el subprograma ficticio se sustituye por el subprograma argumento, concretándose así su acción o resultado (dependiendo de si es un procedimiento o una función, respectivamente).

Cuando el parámetro de un subprograma es otro subprograma, la consistencia entre la llamada y la definición requiere que los subprogramas (parámetros)

formales (`F` y `G`) y los reales (`Sin` y `Cos`) tengan el mismo encabezamiento, esto es, igual número de parámetros y del mismo tipo. Además, en el caso de tratarse de funciones, el tipo del resultado de la función debe ser también el mismo.

La ganancia en flexibilidad es clara: no se ha definido la función máximo, punto a punto, de dos funciones fijas, sino de dos funciones cualesquiera (predefinidas o definidas por el usuario). Así, por ejemplo, si el usuario tiene definida una función `Cubo`, la siguiente es otra llamada válida:

```
WriteLn(MaxFuncs(Abs, Cubo, 2 * x + y))
```

En Pascal, los parámetros-subprograma sólo pueden ser datos de entrada (no de salida). Además, el paso de parámetros-subprograma no puede anidarse.

Al ser esta técnica un aspecto complementario de este libro, simplemente se añaden algunos ejemplos prácticos para mostrar su aplicación. Los ejemplos son el cálculo de la derivada de una función en un punto, la búsqueda dicotómica de una raíz de una función dentro de un cierto intervalo y la transformación de listas.

A.1.1 Ejemplo 1: derivada

La derivada de una función[1] en un punto se puede hallar aproximadamente mediante la expresión

$$\frac{f(x+\Delta x)-f(x)}{\Delta x}$$

tomando un Δx suficientemente pequeño. El cálculo de la derivada se ilustra gráficamente en la figura A.2.

Como este cálculo depende de la función f, el subprograma en Pascal deberá incluir un parámetro al efecto:

```
function Derivada(function F(y: real): real; x: real): real;
  {PreC.:  F debe ser continua y derivable en x}
  {Dev.  el valor aproximado de la derivada de de F en x}
  const
    DeltaX = 10E-6;
begin
  Derivada:= (F(x+DeltaX) - F(x))/DeltaX
end; {Derivada}
```

Si efectuamos una llamada a esta función pasándole la función `Sin`, y la abscisa `Pi/3`, obtenemos un valor aproximado, como era de esperar:

```
Derivada(Sin, Pi/3) ⇝ 4.9999571274E-01
```

[1] La función deberá ser continua y derivable en el punto considerado.

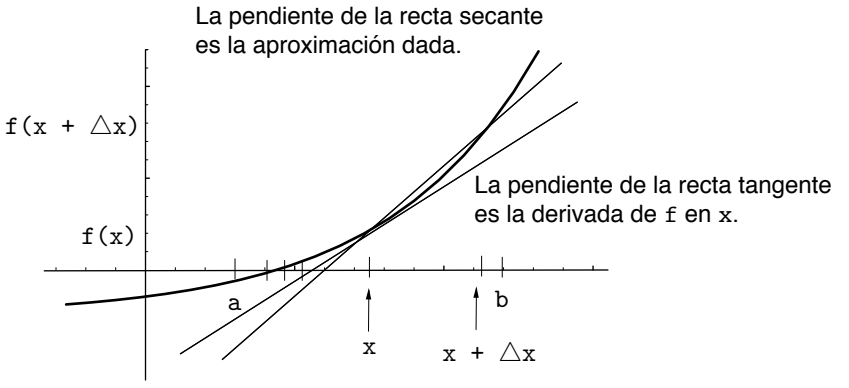


Figura A.2.

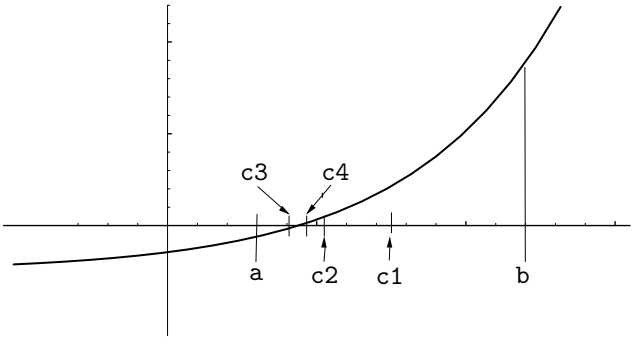


Figura A.3.

A.1.2 Ejemplo 2: bipartición

Este método para calcular los ceros o raíces de una función se expuso en el apartado 6.5.1 para una función particular, y la generalización natural es definirlo en Pascal como un subprograma que opere con una función cualquiera, además de efectuar la búsqueda en un intervalo cualquiera.

Las condiciones exigidas son que la función debe ser continua en ese intervalo y tener distinto signo en sus extremos: digamos para simplificar que deberá ser negativa en el extremo izquierdo y positiva en el derecho, como se muestra en la figura A.3.

Este cálculo puede incluirse dentro de una función `CeroBipar` que reciba como parámetros la función de la que se calcula la raíz y los extremos del intervalo

en el que se busca ésta. La función devuelve la abscisa de la raíz.

Siguiendo las ideas expuestas en el apartado 6.5.1, una primera aproximación al diseño puede ser

```
izda:= extrIz;
dcha:= extrDc;
Reducir el intervalo
Dar el resultado
```

Recordemos que la tarea *reducir el intervalo* se puede refinar mediante un bucle **while**, utilizando una variable adicional `puntoMedio` para almacenar el valor del punto central del intervalo. `F` será el identificador del parámetro que recogerá la función de la que se busca la raíz. El subprograma final es el siguiente:

```
function CeroBipar(function F(x: real): real;
                   extrIz, extrDc: real): real;
  {PreC.: F continua en [extrIz, extrDc],
            F(extrIz) < 0 < F(extrDc) }
  {PostC.: F(CeroBipar(F, extrIz, extrDc)) ≈ 0 ,
            extrIz < CeroBipar(F, extrIz, extrDc) < extrDc }
  const
    Epsilon= error permitido;
  var
    puntoMedio: real;
begin
    {Reducción del intervalo}
  while (extrDc - extrIz) > 2 * Epsilon do begin
      {Inv.:  F(extrIz) < 0 y F(extrDc) > 0 }
    puntoMedio:= (extrDc - extrIz) / 2;
    if F(puntoMedio) < 0 then
        {El cero se encuentra en [puntoMedio,extrDc]}
      extrIz:= puntoMedio
    else {El cero se encuentra en [extrIz,puntoMedio]}
      extrDc:= puntoMedio
  end; {while}
  CeroBipar:= puntoMedio
end. {CeroBipar}
```

Por ejemplo, si suponemos implementado en Pascal el polinomio $p(x) = x^2 - 2$ mediante la función `P`, se obtendrá un cero del mismo en [0,2] efectuando la llamada

```
CeroBipar(P, 0, 2)
```

A.1.3 Ejemplo 3: transformación de listas

Una transformación frecuente de las listas consiste en aplicar cierta función (genérica) a todos sus elementos. Una implementación en Pascal de esta transformación es casi directa desarrollando un procedimiento `AplicarALista` que tenga como argumentos la función a aplicar y la lista a transformar:

```
procedure AplicarALista(function F(x: real): real;
                        var lista: tListaR);
  {Efecto:  aplica F a cada uno de los elementos de lista}
  var
    aux: tListaR;
begin
  aux:= lista;
  while aux <> nil do begin
    aux^.valor:= F(aux^.valor);
    aux:= aux^.siguiente
  end {while}
end; {AplicarALista}
```

Donde los elementos de la lista deben tener un tipo conocido y fijo (en el ejemplo, `real`),

```
type
  tListaR = ^tNodoR;
  tNodoR = record
      valor: real;
      sig: tListaR
  end; {tNodoR}
```

y éste debe ser el mismo antes y después de su transformación, por lo que la función convierte elementos de ese tipo en elementos del mismo. En este ejemplo la lista está formada por valores reales, por lo que también lo son el argumento y el resultado de la función.

Así, por ejemplo, la llamada `AplicarALista(Sqr, listaDeReales)` tendrá por efecto elevar al cuadrado todos los elementos de la lista `listaDeReales`.

A.2 Variables aleatorias

A menudo es necesario construir programas en los que interviene el azar: este comportamiento indeterminista es inevitable con frecuencia (por ejemplo, en procesos de muestreo), y otras veces es deseable (en programas relacionados con simulación, juegos, criptografía, etc.)

En este apartado se introduce el uso del azar para resolver problemas algorítmicamente.[2] Empezaremos por resumir los mecanismos que ofrece Turbo Pascal para introducir el azar en nuestros programas, así como la forma de definir otros, sencillos y útiles; después se verá un método para simular variables aleatorias cualesquiera, para terminar dando algunos ejemplos de aplicación.

A.2.1 Generación de números aleatorios en Turbo Pascal

La construcción de un buen generador de números seudoaleatorios no es una tarea fácil; por eso y por sus importantes aplicaciones, casi todos los lenguajes de programación incluyen algún generador predefinido.

Para obtener los valores aleatorios en Turbo Pascal, se emplea la función `Random`, que genera un valor seudoaleatorio y que se puede utilizar de dos modos distintos:

- Con un argumento entero y positivo[3] `n`, en cuyo caso el resultado es un número extraído uniformemente del conjunto $\{0, \ldots, n-1\}$:

```
Random(6) ⇝ 2
Random(6) ⇝ 5
Random(6) ⇝ 0
```

- Sin argumento, en cuyo caso el resultado es un real, extraído uniformemente del intervalo $[0, \ldots, 1)$:

```
Random ⇝ 0.78593640172
Random ⇝ 0.04816725033
```

Cuando estas funciones se usan en un programa, se debe usar el procedimiento `Randomize` (sin argumentos), para "arrancar" la generación de números aleatorios, haciendo intervenir información cambiante, no controlada, como determinadas variables del sistema (el reloj), de modo que las distintas ejecuciones de ese programa funcionen de diferente modo.

Como ejemplo de uso, damos el siguiente programa, que simula el lanzamiento de un dado y de una moneda:

[2] Hay obviamente una dificultad intrínseca para reproducir el azar (de naturaleza indeterminista) mediante algoritmos (que son procesos deterministas). De hecho, en la práctica sólo se consigue simular un comportamiento "aparentemente" aleatorio –*seudoaleatorio*– aunque esto es suficiente para un gran número de aplicaciones.

[3] En realidad, en Turbo Pascal debe ser de tipo `word` (véase el apartado B.3).

```
Program DadoYMoneda (output);
  const
    N = 10; {núm.  de lanzamientos}
  var
    i, lanzamientos, caras, cruces:  integer;
begin
  Randomize;
  for i:= 1 to N do {N lanzamientos de dado}
    WriteLn(Random(6)+1);
  for i:= 1 to N do {N lanzamientos de moneda}
    if Random < 0.5 then
      WriteLn('Cara')
    else
      WriteLn('Cruz')
end.  {DadoYMoneda}
```

Como puede imaginarse, la función `Random` se puede combinar con otras operaciones y funciones para lograr efectos más variados. Por ejemplo la expresión

$$\texttt{Random} * (\texttt{b} - \texttt{a}) + \texttt{a}$$

produce el efecto de una variable aleatoria uniforme del intervalo $[a, b)$. De modo similar, la expresión

$$\texttt{Random}(\texttt{b} - \texttt{a} + 1) + \texttt{a}$$

produce el efecto de una variable aleatoria uniforme del conjunto $\{a, \ldots, b\}$.

A.2.2 Simulación de variables aleatorias

Aunque las variables aleatorias uniformes bastan en un gran número de situaciones, muchas veces se requiere generar otras variables siguiendo otras funciones de distribución. En este apartado se introduce el método de la *transformada inversa*,[4] que permite generar variables aleatorias arbitrarias. En los ejercicios se describen sin justificar algunos métodos particulares.

En primer lugar se presenta este método para variables aleatorias continuas. Si $F : \mathbb{R} \to [0, 1]$ es una función de distribución cualquiera, el método consiste en generar la variable aleatoria $y \sim Unif[0, 1)$, y hallar el x tal que $F(x) = y$ (véase la figura A.4). En otras palabras, si $y \sim Unif[0, 1)$, entonces $F^{-1}(y) \sim F$.

Por ejemplo, supongamos que deseamos generar una *variable aleatoria uniforme* en el intervalo $[a, b)$. Como su función de distribución es,

$$F(x) = \begin{cases} 0 & \text{si } x \leq a \\ \frac{x-a}{b-a} & \text{si } a \leq x \leq b \\ 1 & \text{si } b < x \end{cases}$$

[4]También conocido como *look up-table*.

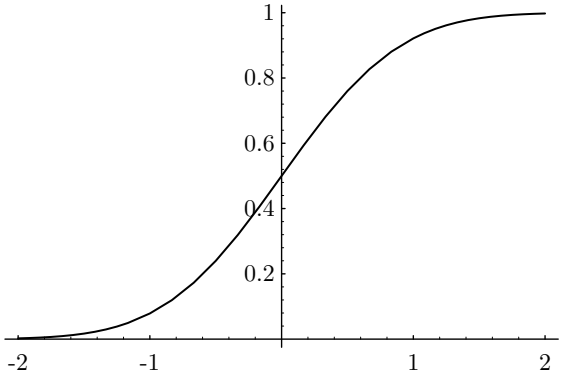


Figura A.4.

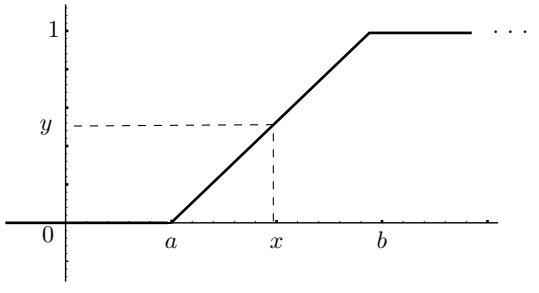


Figura A.5.

para $y \in [0,1)$ se tiene $F^{-1}(y) = a + y * (b-a)$. Por lo tanto, si se genera $y \sim Unif[0,1)$ (mediante `y:= Random` simplemente, por ejemplo), basta con hacer `x:= a+y*(b-a)` para obtener la variable aleatoria $x \sim Unif[a,b)$, como se ve en la figura A.5.

Naturalmente, no siempre es tan fácil invertir F, e incluso a veces es imposible hacerlo por medios analíticos. Sin embargo, la cantidad $x = F^{-1}(y)$, conocida y, siempre puede hallarse como el cero de la función $F(x) - y$ (véase el apartado 6.5.3), de la que sabemos que es decreciente, por lo que es idóneo el método de bipartición (véase el apartado 6.5.1).

El método expuesto es sencillo y se adapta bien a variables discretas. Sea la función de probabilidad $Prob(x = i) = p_i$, para $1 \leq i \leq n$; su función de distribución es $P(k) = \sum_{i=1}^{k} p_i$ para $1 \leq k \leq n$, y expresa la probabilidad de que $x \leq i$. Entonces, el método consiste en generar la variable aleatoria $y \sim Unif[0,1)$, y hallar el mínimo k tal que $P(k) \geq y$ (véase la figura A.6).

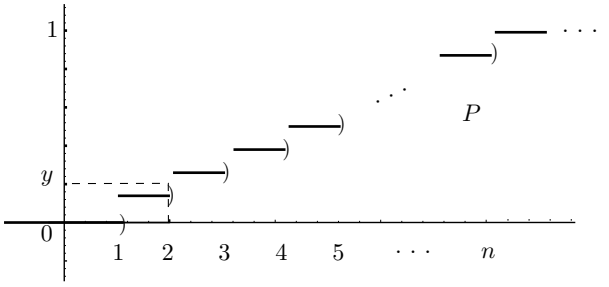


Figura A.6.

Por ejemplo, supongamos que se desea trucar un dado de forma que dé las cantidades 1, ..., 6 con probabilidades 0'15, 0'1, 0'15, 0'15, 0'3, 0'15 respectivamente. Para ello, se hallan las cantidades $P(k)$, que resultan ser 0'15, 0'25, 0'40, 0'55, 0'85, 1'00, respectivamente; luego se genera $y \sim Unif[0,1)$, mediante `y:= Random`, y finalmente, basta con hallar el $mín\{k$ tal que $P(k) \geq y\}$. Si las cantidades $P(k)$ se almacenaron en un array, esta búsqueda se puede realizar por inspección de la tabla,[5] que tiene el siguiente contenido:

$Prob(x = i)$	$P(k)$	i
0'15	0'15	1
0'1	0'25	2
0'15	0'40	3
0'15	0'55	4
0'3	0'85	5
0'15	1'00	6

Si `y:= Random` genera el valor $0'75$, por ejemplo, hay que buscar el menor $P(k) \geq 0'75$, y a partir de él localizar en la tabla de búsqueda la cara del dado que le corresponde (el cinco, en este caso).

La inspección se puede hacer secuencialmente o mediante búsqueda dicotómica (véase el apartado 15.1). El desarrollo se deja como ejercicio (véase el ejercicio 15).

A.2.3 Ejemplos de aplicación

Las aplicaciones de los algoritmos no deterministas son múltiples y variadas. Se muestran en este apartado dos de ellas como botón de muestra, remitiendo

[5] Así se explica su denominación *look up-table* en inglés.

al lector interesado a los ejercicios, donde se indican otras, y a los comentarios bibliográficos.

Como ejemplo inicial, considérese que un supermercado desea hacer un sorteo entre sus clientes, de forma que sus probabilidades de ser premiados sean proporcionales al dinero gastado en la tienda. Trivialmente se ve que esta situación es un caso particular del dado trucado, explicado al final del apartado anterior, por lo que no requiere mayor explicación (véase el ejercicio 15).

Un ejemplo típico es el acto de barajar, que consiste sencillamente en

Dado un array A de n elementos,
Situar en A_1 *uno cualquiera,*
  escogido aleatoriamente entre A_1 *y* A_n*.*
...
Situar en A_{n-1} *uno cualquiera,*
  escogido aleatoriamente entre A_{n-1} *y* A_n*.*

donde la elección entre A_{izda} y A_{dcha} se efectúa mediante la asignación

`posic:=` *variable aleatoria uniforme del conjunto* {*izda, . . . , dcha*}

referida en el apartado A.2.1 y en el ejercicio 12c de este capítulo.

Finalmente, esbozamos el método del *acontecimiento crítico*, para simular una cola (una ventanilla de un banco, por ejemplo), donde llega un cliente cada λ_1 unidades de tiempo y tarda en ser atendido λ_2 unidades de tiempo, por término medio.[6]

Un paso intermedio del diseño de este programa puede ser el siguiente:

```
  {Dar valores iniciales a los datos}
reloj:= 0;
longCola:= 0;
sigLlegada:= 0; {tiempo hasta la siguiente llegada}
sigSalida:= 0; {tiempo hasta la siguiente salida}
repetir sin parar
  if cola vacía o sigLlegada < sigSalida then begin
      {llegada de cliente}
    reloj:= reloj + sigLlegada;
    sigSalida:= sigSalida - sigLlegada;
    sigLlegada:= ExpNeg(λ1);
    longCola:= longCola + 1;
    EscribirSituacion(reloj, longCola)
  end {then}
  else begin
      {salida de cliente}
```

[6]El tiempo entre llegadas (resp. salidas) es una variable aleatoria exponencial negativa de parámetro λ_1 que se supone desarrollada, `ExpNeg(lambda)` (véase el ejercicio 14).

```
      reloj:= reloj + sigSalida;
      sigLlegada:= sigLlegada - sigSalida;
      sigSalida:= ExpNeg(λ2);
      longCola:= longCola - 1;
      EscribirSituacion(reloj, longCola)
    end {else}
  Fin repetir
```

A.3 Ejercicios

1. Desarrolle un programa completo que utilice la función `MaxFun` que aparece en el texto, y trace la gráfica de la función `MaxFun(Sin, Cos, x)` en el fragmento del plano $XY = [0, 3\pi] \times [-0'5, 1'5]$.

2. Halle un cero del polinomio $p(x) = x^2 - 2$ en $[0, 2]$, siguiendo el proceso indicado en el apartado A.1.2.

3. Aplique la función `Derivada` en un programa al polinomio del ejercicio anterior.

   Desarrolle un programa para comprobar la diferencia entre el resultado obtenido por el programa y el calculado de forma analítica, $p'(x) = 2x$, confeccionando una tabla con los valores de ambos, para diferentes valores de la variable x (por ejemplo, $x \in \{0'0, 0'1, \ldots, 1'0\}$) y de la constante `DeltaX` (por ejemplo, $\{1, 0'5, 0'25, 0'125, 0'0625\}$).

4. Desarrolle un subprograma para hallar el cero de una función dada siguiendo el método de Newton-Raphson explicado en el apartado 6.5.2.

5. Desarrolle un subprograma que halle $\sum_{i=inf}^{sup} a_i$, para una sucesión cualquiera $a : \mathbb{N} \to \mathbb{R}$.

   Aplíquese al cálculo de $\sum_{i=1}^{n} \frac{1}{i!}$, siendo n un dato dado por el usuario.

6. Desarrolle un subprograma que halle $\int_a^b f(x)dx$, siguiendo los métodos siguientes:

   (a) iterativamente, mediante la correspondiente descomposición en rectángulos (véase el ejercicio 11 del capítulo 6).

   (b) recursivamente (véase el ejercicio 7 del capítulo 10).

7. Desarrolle una nueva versión del subprograma `AplicarALista` del apartado A.1.3, siguiendo una estrategia recursiva.

8. Construya un procedimiento `AplicarAArbol` que aplique una función a los elementos de un árbol, transformándolos, de la misma forma que `AplicarALista` lo hace con las listas.

9. Desarrolle un subprograma que filtre los datos de una lista según una función lógica, eliminando los que no cumplan ese test. Si, por ejemplo, se trata de una lista de enteros y el test es la función `Odd`, el efecto del filtro consiste en suprimir los pares.

10. Desarrolle una versión recursiva del subprograma del ejercicio anterior.

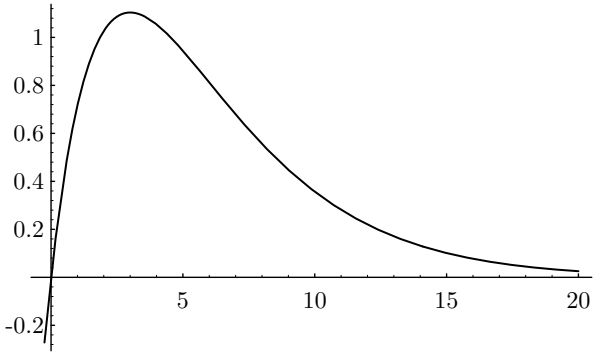


Figura A.7.

11. Construya un procedimiento general de ordenación en el que el criterio por el que se ordenan los elementos sea un subprograma que se suministra como parámetro. El encabezamiento del procedimiento es

```
type
  tVector = array [1..n] of tElemento;
procedure Ordenar(var v:tVector;
                  function EsAnterior(x, y:tElemento):boolean);
```

Con este procedimiento se consigue una gran flexibilidad, al poder ordenar el vector en la forma que más interese al usuario, con llamadas a `Ordenar` en las que `EsAnterior` se particularice con las relaciones de orden "<", OrdenAlfabético, MejorCalificación, etc.

12. Defina funciones aleatorias para simular lo siguiente:

    (a) Una variable aleatoria del intervalo $[a, b)$.

    (b) Una variable aleatoria del intervalo $(a, b]$.

    (c) Una variable aleatoria del conjunto $\{a, \ldots, b\}$.

    (d) Un dado que da un seis con probabilidad 1/2, y un número del uno al cinco con probabilidad uniforme.

    (e) Una moneda trucada, que cae de cara dos de cada tres veces.

13. Genere números naturales de forma que aparezca un 1 con probabilidad 1/2, un 2 con probabilidad 1/4, un 3 con probabilidad 1/8, etc.

14. Empleando el método de la transformada inversa, genere una variable aleatoria exponencial negativa de parámetro λ, cuya función de densidad f viene dada por $f(x) = \lambda e^{-\lambda x}$, y que se representa gráficamente en la figura A.7).

    Se recuerda que, previamente, debe hallarse la función F de distribución correspondiente.

15. Mediante el método de la transformada inversa, desarrolle un dado trucado que dé las cantidades 1 a 6 con probabilidades arbitrarias dadas como dato.

16. Defina una función que genere una variable aleatoria a partir de una función de distribución arbitraria F definida sobre el intervalo $[a, b]$.

A.4 Referencias bibliográficas

Este apartado se ha centrado en el uso del azar, suponiendo que las herramientas necesarias están incorporadas en nuestro lenguaje de programación, lo que es el caso habitual. Sin embargo, el desarrollo de buenos generadores de números aleatorios no es una tarea fácil, e incluso es frecuente encontrar en el mercado y en libros de texto generadores con defectos importantes. En [Yak77, PM88, Dan89, War92], entre otras referencias, se estudian los principios de su diseño, explicando el generador más difundido actualmente (el de Lehmer) y examinando algunos de los errores de diseño más frecuentes; en [War92, Sed88] se describen dos pruebas de calidad (el test espectral y el de la chi-cuadrado).

La simulación de variables aleatorias no uniformes se estudia en multitud de textos. El método de la transformada inversa se puede estudiar en [Yak77, Mor84, PV87], entre otros textos, junto con otros métodos generales (el del "rechazo" y el de la "composición"). Algunos métodos particulares para la simulación de la normal se pueden encontrar en las dos últimas referencias. En [Dew85a] se vincula la simulación de variables aleatorias con algunas aplicaciones de los algoritmos no deterministas a través de amenos ejemplos y situaciones. De esta última referencia se ha tomado el método del acontecimiento crítico, y otras aplicaciones prácticas de la simulación se pueden encontrar en [PV87].

Finalmente, debemos indicar que en el apartado 20.5 se introducen otras aplicaciones de los algoritmos no deterministas.

En este apéndice hemos tocado muy superficialmente la posibilidad de que los parámetros de los subprogramas sean subprogramas a su vez. Esta posibilidad es ampliamente explotada en otros modelos de programación como el funcional, por lo que el lector interesado puede consultar cualquer referencia sobre el mismo, por ejemplo [BW89]. Los cierto es que, en programación imperativa, este mecanismo se usa en muy contadas ocasiones.

Apéndice B

El lenguaje Turbo Pascal

El lenguaje Turbo Pascal posee numerosas extensiones con respecto al lenguaje Pascal estándar, que, por una parte, le confieren una mayor potencia y capacidad, pero por otra merman la posibilidad de transportar sus programas a otros computadores.

Es interesante conocer estas extensiones por las siguientes razones:

- Porque amplían la capacidad para manejar otros tipos numéricos del lenguaje Pascal, superando las limitaciones de los tipos estándar y facilitando el intercambio de este tipo de valores con programas escritos en otros lenguajes.

- Porque existen en muchos otros lenguajes de programación, y por ello han pasado a ser algo admitido y utilizado, siendo un estándar de facto. Por ejemplo, el tipo cadena (*string*) con sus operaciones asociadas.

- Porque son imprescindibles para la utilización de ciertos tipos de datos del Pascal estándar, como ocurre con los archivos, en los que la conexión con su implementación física no está definida en Pascal.

- Porque permiten reforzar ciertas características deseables en todo lenguaje evolucionado. La modularidad de Pascal se refuerza mediante las unidades que nos permiten definir, por ejemplo, los tipos abstractos de datos (véase el capítulo 19), aunque con limitaciones con respecto a otros lenguajes como Modula2.

En los próximos apartados se explican brevemente las particularidades más interesantes de Turbo Pascal, siguiendo el orden en el que se han presentado en el texto los aspectos del lenguaje con los que se relacionan.

B.1 Elementos léxicos

En primer lugar estudiaremos las diferencias más significativas en la forma de escribir los identificadores y ciertos símbolos especiales.

La longitud de los identificadores en Turbo Pascal sólo es significativa en sus 64 primeros caracteres, mientras que en Pascal son significativos todos los caracteres.[1]

En Turbo Pascal el signo `@` es un operador diferente de `^`, mientras que en Pascal se pueden usar indistintamente. El signo `@` en Turbo Pascal permite que un puntero señale a una variable existente no creada como referente del puntero.

En Turbo Pascal un comentario debe comenzar y terminar con el mismo par de símbolos { y } o `(*` y `*)`. En Pascal, un símbolo de un tipo puede cerrarse con el del otro.

B.2 Estructura del programa

El encabezamiento del programa en Turbo Pascal es opcional, por lo que puede omitirse en su totalidad. Sin embargo, si se escribe la palabra **program** deberá ir acompañada del identificador de programa. Los nombres de archivo como `input` y `output` también son opcionales, pero si se escriben, deberá hacerse correctamente.

Las diferentes secciones de declaraciones y definiciones se abren con las palabras reservadas correspondientes como en Pascal estándar. No obstante, en Turbo Pascal se puede alterar el orden de las diferentes secciones y abrirlas repetidas veces.

B.3 Datos numéricos enteros

Turbo Pascal introduce dos tipos numéricos naturales predefinidos llamados `byte` y `word`. Sus dominios incluyen solamente valores enteros positivos, siendo para el tipo `byte` $\{0, ..., 255\}$ y para el tipo `word` $\{0, ..., 65535\}$.

Un valor perteneciente al tipo `byte` ocupa precisamente un byte en memoria (de aquí su nombre), mientras que uno del tipo `word` ocupa dos bytes.

El tipo entero `integer` se complementa con dos nuevos tipos predefinidos denominados `shortInt` y `longInt`. Sus dominios son enteros positivos y nega-

[1] Aunque en el *User Manual & Report* (segunda edición, pg. 9) [JW85] se dice textualmente: *Implementations of Standard Pascal will always recognize the first 8 characters of an identifier as significant.*

tivos, siendo para el tipo `shortInt` $\{-128, ..., 127\}$, mientras que el de `longInt` es $\{-214483648, ..., 2147483647\}$.

El tipo `shortInt` ocupa un byte, el tipo `integer` ocupa 2 y el `longInt` 4 bytes.

Estos tipos son especialmente útiles para realizar cálculos enteros con valores grandes, como el cálculo del factorial, para los que el tipo `integer` resulta muy limitado.

A estos tipos numéricos se les pueden aplicar los mismos operadores y operaciones que al tipo entero, e incluso se pueden asignar entre sí siempre que los valores asignados estén comprendidos dentro de los respectivos dominios.

Existe un tercer tipo llamado `comp` que es un híbrido entre entero y real: se almacena en la memoria como un entero (en complemento a dos), pero se escribe como un real (en notación científica). A pesar de su implementación como entero no es un tipo ordinal. Dado que no se le pueden aplicar las operaciones de tipo entero, lo consideraremos y utilizaremos como real. Se utiliza en aquellas aplicaciones que necesiten valores "grandes", con precisión entera, pero donde no haya que aplicar operaciones enteras.

En Turbo Pascal la expresión `n` **mod** `m` se calcula como `n - (n div m) * m` y no produce error si `m` es negativo, mientras que en Pascal estándar no está definido si `m` es cero o negativo.

B.4 Datos numéricos reales

Turbo Pascal dispone de tres tipos de datos reales (codificados en punto flotante) que complementan al tipo estándar `real`, denominados `single`, `double` y `extended`, además del ya comentado `comp`. Sus diferentes características se muestran en la siguiente tabla:

Tipo	Dominio	Cifras significativas	Ocupación de memoria
`single`	$\{\pm 1.5E-45, ..., \pm 3.4E38\}$	7 u 8	4
`double`	$\{\pm 5.05E-324, ..., \pm 1.7E308\}$	15 ó 16	8
`extended`	$\{\pm 1.9E-4951, ..., \pm 1.1E4932\}$	19 ó 20	10
`comp`	$\{-2^{63}, ..., 2^{63}-1\}$	19 ó 20	8
`real`	$\{\pm 2.9E-39, ..., \pm 1.7E38\}$	11 ó 12	6

Los tipos `single` y `double` cumplen el estándar IEEE 754, que es el más utilizado en representación en punto flotante, lo que los hace idóneos para el intercambio de datos reales con programas escritos en otros lenguajes. Es curioso que el tipo estándar `real` de Turbo Pascal no sea estándar en su codificación interna.

Los tipos reales adicionales, incluyendo el tipo `comp`, admiten todos los operadores y operaciones del tipo `real`, e incluso son asignables entre sí, dentro de sus respectivos dominios.[2]

B.5 Cadenas de caracteres

Es muy frecuente construir programas que precisen cadenas de caracteres para formar nombres, frases, líneas de texto, etc.

En Pascal estándar, este tipo de datos hay que definirlo como un array de caracteres con una longitud fija. Si la secuencia de caracteres tiene una longitud menor que la longitud del array, la parte final de éste queda indefinido. Con el fin de evitar posibles errores, es conveniente almacenar la longitud utilizada del array para no acceder a la parte sin definir.

En Turbo Pascal existe un tipo de datos específico predefinido llamado *string*, que podemos traducir como cadena de caracteres. Este tipo es similar a un array de caracteres, pero su longitud es gestionada automáticamente por el compilador, hasta un cierto límite. Además Turbo Pascal dispone de las funciones y procedimientos necesarios para procesar las cadenas.

B.5.1 Declaración de cadenas

En la declaración de una variable de cadena se define la longitud máxima de la cadena, lo que se conoce como su longitud física.

```
var
  cadena:  string[20];
```

Con esta declaración la variable `cadena` podrá tener a lo sumo 20 caracteres, es decir, este tamaño es un límite máximo, ya que la cadena puede tener menos. Para saber cuántos tiene en realidad, junto con la cadena se guarda un índice que contiene la longitud real de la cadena, lo que se denomina su longitud lógica.

Si al leer la variable `cadena` con la instrucción:

```
ReadLn(cadena)
```

[2]Para su utilización debe estar activada la opción

[Options][Compiler][Numericprocessing][X]8087/80287

(véase el apartado C.3.3).

o asignar un valor con:

```
cadena:= 'Lupirino'
```

escribimos menos de 20 caracteres, el índice almacenará el número real de caracteres escritos. Si escribimos 20 o más, el índice valdría 20, pero en el último caso se perderían los caracteres posteriores al vigésimo, truncándose la cadena.

También es posible declarar su longitud máxima, en cuyo caso la cadena toma una longitud física de 255 caracteres. Por ejemplo:

```
var
  nombre:  string;
```

Se puede acceder a los caracteres de una cadena por sus índices, como en un array, siendo el elemento de índice 0 el que almacena la longitud lógica de la cadena. Por ejemplo, mediante las siguientes instrucciones:

```
longitud:= Ord(cadena[0])
inicial:= cadena[1]
```

se asigna a la variable `longitud` la longitud de la cadena y el primer carácter de ésta a la variable `inicial` .

B.5.2 Operadores de cadenas

Dos cadenas se pueden comparar entre sí con los operadores usuales de relación, considerándose "menor" (anterior) aquella cadena que precede a la otra por orden alfabético.[3] Es decir, la comparación se realiza carácter a carácter, hasta que una cadena difiera de la otra en uno, siendo menor aquélla que tiene menor ordinal en el carácter diferenciador. Puede suceder que una de las dos cadenas termine sin haber encontrado un carácter distinto, en cuyo caso la cadena más corta se considera la menor.

Además de los operadores de relación, se puede aplicar el operador `+`, llamado de concatenación, que une dos cadenas para formar otra. Por ejemplo, haciendo:

```
cadena1:= 'Pepe';
cadena2:= 'Luis';
cadena:= cadena1 + cadena2
```

la variable `cadena` tendría el valor `'PepeLuis'`.

[3]Lamentablemente, el orden entre cadenas se basa en el código ASCII, por lo que no funciona del todo de acuerdo con el orden alfabético español (acentos, eñes). Además, las mayúsculas preceden a las minúsculas, por lo que no pueden compararse entre sí de acuerdo con el orden alfabético.

B.5.3 Funciones de cadenas

Turbo Pascal proporciona un conjunto de funciones predefinidas para procesar las cadenas. Estas funciones se presentan a continuación:[4]

- `Concat`: $\mathcal{S} \times \mathcal{S} \times \ldots \times \mathcal{S} \rightarrow \mathcal{S}$.

  Concatena las cadenas argumento para producir una cadena resultado. Produce el mismo resultado que el operador `+`.

- `Length`: $\mathcal{S} \rightarrow \mathcal{Z}$.

  Halla la longitud lógica (entero) de la cadena argumento. La función `Length` equivale al ordinal del carácter 0 de la cadena argumento.

- `Pos` $(\mathcal{S}_1, \mathcal{S}_2) \rightarrow \mathcal{Z}$.

  Indica la posición (un entero) de la primera cadena dentro de la segunda, o el valor 0 si no se encuentra. Esta función es especialmente útil para buscar un texto dentro de otro.

- `Copy(s: string; z1, z2: integer): string;`

  Extrae de `s` una subcadena formada por `z2` caracteres a partir del `z1`-ésimo (incluido)

Veamos algunos ejemplos, con sus salidas:

```
var
 cadena1, cadena2, cadena3:   string[40];
 ...
 cadena1:= 'Alg.';                          ||
 cadena2:= ' y estr.  de datos';            ||
 cadena3:= Concat(cadena1, cadena2);        ||
 WriteLn(cadena3);                          || Alg. y estr. de datos
 WriteLn(Length(cadena3));                  || 21
 WriteLn(Pos(cadena2, cadena3));            || 5
 WriteLn(Copy(cadena3, 8 ,4))               || estr
```

B.5.4 Procedimientos de cadenas

Como complemento de las funciones predefinidas, Turbo Pascal también dispone de un conjunto de procedimientos de gran utilidad para el manejo de cadenas. Estos procedimientos son los siguientes:

[4]Dada la diversidad de tipos de algunos procedimientos y funciones de cadenas, se ha optado por dar sus encabezamientos en lugar de su definición funcional.

- `Delete(var s: string; z1, z2: integer)`

  Borra `z2` caracteres de la cadena `s` a partir del `z1`-ésimo (incluido). Al utilizarlo, la cadena reduce su longitud en el número de caracteres eliminados.

- `Insert(s1: string; var s2: string; z: integer)`

  Inserta en `s2` la cadena `s1` a partir de la posición `z`. El procedimiento `Insert`, por el contrario, aumenta la longitud de la cadena en el número de caracteres insertados.

- `Str(r: real ; var s: string)`

  Convierte el valor real `r` (también puede ser un entero `z`) en la cadena `s`. `Str` convierte un valor numérico en su representación como cadena de caracteres, lo que permite, por ejemplo, medir la longitud en caracteres de un número. Se utiliza también en aplicaciones gráficas, donde es obligatorio escribir cadenas.

- `Val(s: string; var r: real; var z: integer)`

  Convierte la cadena `s` en el valor real `r` (también puede ser un entero) y devuelve un código entero `z`, que es 0 si se puede hacer la conversión, y en caso contrario señala la posición del error. Este procedimiento es quizás el más interesante de todos, al efectuar la conversión de una cadena formada por dígitos y aquellos símbolos permitidos para formar un número (tales como la letra E mayúscula o minúscula y los símbolos punto, más y menos) en su valor numérico. Si por error la cadena no tiene forma correcta de número, se señala la posición del carácter causante del fallo, lo que permite corregirlo. De esta forma se puede robustecer el proceso de introducción de números, que es una de las causas principales de errores de ejecución.

Veamos algunos ejemplos, tomando como punto de partida las asignaciones del ejemplo anterior.

```
Delete(cadena3, 11, 14);
WriteLn(cadena3);                    | Algoritmos de datos
Insert(' simples', cadena3, 20);
WriteLn(cadena3);                    | Algoritmos de datos simples
valNum1:= 123;
Str(valNum1, cadena1);
WriteLn(cadena1);                    | 123
Val('12A', valNum1, error);
WriteLn(valNum1,' ',error);          | 0 3
Val(cadena1, valNum1, error);
WriteLn(valNum1,' ',error)           | 123 0
```

B.6 Tipos de datos estructurados

Las particularidades de Turbo Pascal en cuanto a estos tipos de datos son escasas: en lo referente a arrays, se debe señalar que los procedimientos `Pack` y `Unpack` no están definidos en Turbo Pascal.

B.7 Instrucciones estructuradas

Las diferencias con Pascal en cuanto a instrucciones estructuradas se limitan al tratamiento de la instrucción **case**, que en Turbo Pascal presenta tres variaciones, que son las siguientes:

- En Turbo Pascal el valor de la expresión selectora de un **case** puede no coincidir con alguna de sus constantes sin que se produzca error, como sucedería en Pascal estándar. En este caso, en Turbo Pascal continúa la ejecución del programa en la instrucción que sigue a **case**.

- La instrucción **case** en Turbo Pascal puede disponer de una parte **else** que se ejecuta cuando la expresión selectora no coincide con ninguna de sus constantes.

- Dentro de las constantes de la instrucción **case**, Turbo Pascal permite la definición de intervalos, lo que no está permitido en Pascal estándar.

A continuación se muestra un pequeño ejemplo de las diferencias anteriores y su sintaxis.

```
program ClasificacionDeCaracteres (input, output);
  var
    car:  char;
begin
  Write('Introduzca un carácter:  ');
  ReadLn(car);
  case car of
    'a'..'z':WriteLn('Es minúscula');
    'A'..'Z':WriteLn('Es mayúscula');
    '0'..'9':WriteLn('Es número')
    else WriteLn('Es un símbolo')
  end {case}
end. {ClasificacionDeCaracteres}
```

B.8 Paso de subprogramas como parámetros

Turbo Pascal difiere de Pascal estándar en la forma de realizar la declaración y paso de subprogramas como parámetros. En Turbo Pascal los subprogramas deben ser obligatoriamente de un tipo con nombre para poder ser pasados como parámetros.

Por ejemplo, en la definición de la función `Derivada` (véase el apartado A.1.1) se utilizaba como parámetro una función de argumento real y resultado también real. El tipo de esta función se define en Turbo Pascal como se muestra a continuación:

```
type
  tMatFun = function (x:  real):  real;
```

Para un procedimiento con dos parámetros enteros se escribiría:

```
type
  tProcInt = procedure (a, b:  integer);
```

Los identificadores utilizados en estos encabezamientos se utilizan a efectos de la definición del tipo sin que tengan ninguna repercusión sobre el resto del programa.

Para declarar la función como parámetro formal, la declaramos del tipo `tMatFun` dentro del encabezamiento de la función ejemplo. De esta forma:

```
function Derivada (Fun:  tMatFun; x:  real):  real;
  const
    DeltaX = 10E-6;
begin
  Derivada:= (Fun(x + DeltaX) - Fun(x))/DeltaX
end; {Derivada}
```

Dentro del programa principal efectuamos la llamada a la función `Derivada` pasándole cualquier función definida por el usuario que encaje con el tipo `tMatFun`. Suponiendo definida una función `Potencia`, una posible llamada sería:

```
WriteLn('La derivada es:  ', Derivada(Potencia, x))
```

La utilización de subprogramas como parámetros requiere, dentro del esquema de gestión de memoria de Turbo Pascal, la realización de llamadas fuera del segmento de memoria donde reside el programa. Estas *llamadas lejanas* se activan marcando la opción

`[F10] [Options] [Compiler] [Code generation] [X] [Force far calls]`

dentro del entorno integrado (véase el apartado C.3.3). También se puede hacer desde el programa fuente utilizando las denominadas *directrices de compilación*, que son unas marcas que delimitan la parte del programa que debe compilarse con esta u otras funciones activadas. En el caso de las llamadas lejanas, la directriz de activación es {`$F+`} y la desactivación es {`$F-`}.

B.9 Archivos

Para poder trabajar con archivos en Turbo Pascal es necesario relacionar el archivo externo, existente en el dispositivo de E/S y nombrado según el sistema operativo, con la variable de tipo archivo definido en nuestro programa. Una vez establecida esta relación se puede proceder a utilizar las instrucciones `Reset` o `ReWrite` y posteriormente las de lectura o escritura respectivamente, que tendrán efecto sobre el archivo externo relacionado.

Esta conexión se establece mediante el procedimiento `Assign` de Turbo Pascal que tiene como parámetros el identificador de la variable archivo y una cadena que representa el nombre del archivo externo expresado en el lenguaje de comandos del sistema operativo. Por ejemplo, la instrucción:

```
var
  archivo:  text;
  ...
  Assign(archivo, 'C:\CARTAS\CARGEST.DOC')
```

vincula la variable `archivo` con el archivo llamado `CARGEST.DOC` existente en la unidad `C:`, directorio `CARTAS`.[5]

La otra instrucción adicional de Turbo Pascal para el proceso de archivos es `Close`:

```
Close(archivo)
```

que sirve para cerrar un archivo una vez que se termina de procesarlo. Al ejecutarse la instrucción `Close`, aquellas operaciones de lectura o escritura que pudieran quedar pendientes son completadas, quedando el archivo externo actualizado. A continuación el archivo argumento es cerrado y se pierde la conexión establecida por `Assign`.

[5]Si se omite la unidad se tomará la actual, al igual que si se omiten los directorios. En cualquier caso deberá figurar un nombre de archivo, no estando permitido el uso de comodines (véase el apartado A.1.2 de [PAO94]).

En Turbo Pascal las variables de archivo no tienen asociada la variable intermedia (cursor) definida por el operador ^ propia de Pascal estándar. Al escribir el símbolo ^ detrás de un variable de archivo se produce un error. Esta diferencia es importante, pues impide que en Turbo Pascal se pueda inspeccionar una componente del archivo sin leerla. Por esta razón ciertos programas que emplean este tipo de acceso deben ser modificados.

Los procedimientos `Get` y `Put`, para el manejo de archivos, tampoco están definidos en Turbo Pascal. Aquellos programas que los utilizan deben modificarse.

En Turbo Pascal la lectura de una marca de fin de línea, en un archivo de texto, devuelve el carácter ASCII 13 (retorno de carro), y si continúa la lectura, el caracter ASCII 10 (alimentación de línea). En Pascal estándar la lectura de una marca de fín de línea se realiza como la de un único carácter y devuelve un espacio en blanco.

Turbo Pascal dispone de numerosas extensiones para el tratamiento de archivos. Algunas realizan llamadas a las funciones del sistema operativo permitiendo, por ejemplo, cambiar el directorio de trabajo, borrar un archivo, etc. Otras permiten efectuar un acceso directo a las componentes de los archivos mejorando el tratamiento secuencial de Pascal estándar. También se permiten ficheros sin tipo para realizar operaciones a bajo nivel. Para su estudio remitimos a la bibliografía complementaria.

B.10 Memoria dinámica

Las diferencias en cuanto al manejo de memoria dinámica residen en que en Turbo Pascal los procedimientos `New` y `Dispose` sólo reciben una variable de tipo puntero, mientras que en Pascal estándar se permiten parámetros adicionales. Recordemos, además, que en Turbo Pascal el operador `@` tiene un significado diferente de ^ (véase el apartado B.1).

B.11 Unidades

Las unidades consisten en conjuntos de objetos, tales como constantes, tipos, variables, procedimientos y funciones que pueden ser definidos o declarados e incluso iniciados en la propia unidad. Estos objetos normalmente están relacionados entre sí y se orientan a la resolución de ciertos problemas o tareas.

Con las unidades se puede ampliar el repertorio de instrucciones del lenguaje de una forma modular, agrupándolas por acciones, sin tener que mostrar cómo se realizan estas acciones, que quedan ocultas en una parte privada.

Hay dos tipos de unidades en Turbo Pascal, aunque ambas son utilizadas de idéntica forma. Estos dos tipos son:

- Unidades predefinidas, que se dedican a tareas concretas como, por ejemplo, la interacción con el sistema operativo, el tratamiento de la pantalla de texto o la creación de gráficos.
- Unidades definidas por el programador, para resolver otros problemas no previstos en las unidades predefinidas.

Cada unidad es compilada por separado y es incorporada a un programa mediante una llamada a la misma, realizada al comienzo del programa (antes de las definiciones y declaraciones), en una claúsula

uses *unidad*

Las unidades se incorporan a los programas, que, de esta forma, pueden acceder a los objetos que forman la unidad y utilizarlos en la resolución de dichos problemas o tareas.

Las unidades son una forma apropiada (en Turbo Pascal) para construir bibliotecas de subprogramas para realizar cálculos o procesos concretos. También se utilizan para la definición de tipos abstractos de datos (véase el capítulo 19).

B.11.1 Unidades predefinidas de Turbo Pascal

El lenguaje Turbo Pascal incorpora un conjunto de unidades que le dan una mayor potencia y flexibilidad. Son las siguientes:

- `System`: Esta unidad incluye todas las instrucciones predefinidas de Pascal estandar. Es incorporada de forma automática en todos los programas, por lo que no es necesario nombrarla en la cláusula **uses**.
- `DOS`: En esta unidad se pueden encontrar los equivalentes en Pascal de las principales llamadas al sistema operativo.
- `Crt`: Contiene funciones y procedimientos para trabajar con la pantalla de texto.
- `Printer`: Es una pequeña unidad que facilita el trabajo con la impresora. En ella se trata a la impresora como un archivo de texto llamado `lst`. Un procedimiento `Write` o `WriteLn` que se dirija al archivo `lst`, tendrá como efecto el envío de la salida a la impresora. Veamos un ejemplo:

```
uses printer;
...
WriteLn (lst, 'texto')
```

- `Graph3`: Es una unidad para la compatibilidad con los gráficos de tortuga[6] de la versión 3.0 de Turbo Pascal. Depende de la unidad `Crt`, por lo que ésta debe ser llamada previamente.

- `Turbo3`: Es una unidad para compatibilidad con ciertas instrucciones de la versión 3.0. Al igual que `Graph3` también depende de `Crt`.

- `Graph`: Es la unidad donde se definen las rutinas gráficas necesarias para usar la pantalla en los modos gráficos de alta resolución.

Los contenidos particulares de cada una de ellas pueden consultarse en la bibliografía complementaria.

B.11.2 Unidades definidas por el usuario

De la misma forma que al escribir nuevos procedimientos y funciones un programador amplía las ya existentes en el lenguaje y puede utilizarlas en su programa, se pueden escribir nuevas unidades, y añadirlas a las existentes en Turbo Pascal. Una vez compiladas se pueden incorporar a aquellos programas que las necesiten sólo con nombrarlas en la cláusula **uses**.

Una de las principales aplicaciones de las unidades definidas por el usuario es la creación de nuevos tipos de datos complejos, cuyas definiciones y operaciones asociadas a los mismos son incluidas en una unidad. Estos tipos abstractos de datos pueden incorporarse en la forma descrita anteriormente a aquellos programas que los precisen (véase el capítulo 19).

Las unidades tienen dos partes: una pública, llamada **interface**, donde se definen los objetos que la unidad ofrece para que puedan ser usados por los programas, y otra privada llamada **implementation**, donde se concretan y desarrollan los objetos mencionados en la parte pública y donde se definen otros objetos locales privados que quedan ocultos y a los que no es posible acceder desde los programas que utilicen la unidad.

Para escribir una unidad tenemos que conocer su estructura y ésta es:

[6]Los gráficos de tortuga fueron desarrollados por Seymour Papert en el MIT dentro del lenguaje Logo y consisten en un paradigma de una tortuga que se desplaza un cierto número de pasos en una dirección o que gira un cierto ángulo, y que va dejando un rastro que forma el gráfico.

unit *nombre de la unidad*;
interface
  uses *lista de unidades*;
  definiciones y declaraciones públicas;
implementation
  definiciones y declaraciones privadas;
  procedimientos y funciones;
begin
  código de iniciación
end.

En primer lugar aparece la palabra reservada **unit**, seguida por el nombre de la unidad, de forma similar al nombre de un programa.

La palabra reservada **interface** abre las definiciones y declaraciones públicas. Si la unidad en cuestión depende de otras unidades, debe situarse en primer lugar la cláusula **uses** seguida por la lista de unidades necesitadas. A continuación se deben definir constantes, tipos, y declarar variables, y los encabezamientos de procedimientos y funciones que serán visibles al programa que utilice la unidad. Los cuerpos de los procedimientos y funciones declarados no se incluyen aquí.

La palabra reservada **implementation** inicia la parte privada, en la que deben desarrollarse los procedimientos y funciones cuyos encabezamientos se han declarado en la parte **interface**. Para ello deben repetirse en esta parte los encabezamientos, si bien pueden abreviarse eliminando sus parámetros si los tienen.

La parte **implementation** puede completarse con otros objetos enteramente privados, incluso otros procedimientos y funciones, que pueden ser utilizados por los públicos pero que no queremos que sean visibles. Estos subprogramas deberán tener su encabezamiento completo. Todos los objetos definidos o declarados en la parte de **interface** son visibles en **implementation**.

Después de esta parte se puede ubicar lo que se denomina código de iniciación, que consiste en un conjunto de instrucciones para dar valores iniciales a aquellas estructuras variables utilizadas por la propia unidad, en este caso se coloca un **begin**, como se mostró en el esquema anterior.

En el capítulo 19 pueden encontrarse ejemplos de definición y utilización de unidades dedicadas a tipos abstractos de datos.

Una vez escrita y depurada la unidad, ésta se compila dando lugar a un archivo con extensión `TPU`. Cuando se compila un programa que contiene una cláusula **uses** seguida por el nombre de la unidad, el compilador busca el archivo `*.TPU` correspondiente, agrega sus definiciones y declaraciones a las del propio programa, y enlaza el código de la unidad y del programa. Dado que la unidad ha sido compilada previamente, la conexión entre ambos es bastante rápida.

B.11.3 Modularidad incompleta de Turbo Pascal

La utilización de unidades en Turbo Pascal refuerza los aspectos modulares del lenguaje Pascal estándar siendo equivalentes, con pequeñas limitaciones, a los módulos existentes en otros lenguajes.

Las unidades permiten solucionar ciertos problemas de jerarquía modular como, por ejemplo, las llamadas a subprogramas desde otros varios, lo que obligaba a situar los subprogramas llamados por encima de su verdadero nivel para hacerlos accesibles a dos o más subprogramas diferentes. La solución a este problema se alcanza incorporando una unidad con los subprogramas llamados.

Las unidades tiene una modularidad de acciones completa, al estar separadas las partes pública y privada de los subprogramas, lo que les permite alcanzar una verdadera ocultación de la información. Sin embargo, la modularidad de los datos no es completa, al no permitir mencionar públicamente tipos con una implementación privada (oculta) como en otros lenguajes, por ejemplo, Modula2 o Ada.

Por ello, cuando utilizamos las unidades de Turbo Pascal para la definición de tipos abstractos de datos, su declaración y definición tienen que estar en la parte pública **interface**.

Apéndice C

El entorno integrado de desarrollo

Turbo Pascal© es un producto comercial desarrollado por la empresa Borland International, Inc., cuyo uso está muy extendido.[1]

Turbo Pascal ha ido evolucionando a lo largo del tiempo de acuerdo con las necesidades del mercado. Esta evolución se ha concretado en la aparición de sucesivas versiones del producto. Los principales saltos cualitativos se producen en el paso de la versión 3.0 a la 4.0, al introducirse un entorno integrado de desarrollo, y en el paso de la versión 5.0 a la 5.5 permitiendo algunas características de la programación orientada a objetos. La evolución posterior tiende a completar las posibilidades del entorno y facilitar su utilización (uso del ratón en la versión 6.0, resaltado de palabras reservadas en la 7.0, etcetera). La versión más reciente en el momento de escribir estas líneas es la 7.01, que contiene la ayuda traducida al castellano. Además, existe una versión para Windows© denominada Delphi© .

El contenido de este apéndice corresponde a la versión 7.0, si bien se puede aplicar con muy pequeñas variaciones desde la versión 4.0.

C.1 Descripción del entorno

Turbo Pascal no es sólo un compilador de un lenguaje de programación, sino un completo entorno integrado de desarrollo compuesto por todos los componentes necesarios para desarrollar programas, entre otros:

- Un potente *editor*, que permite escribir y modificar programas (y texto en general), con la posibilidad de cortar, copiar, pegar, buscar y reemplazar texto.

[1] El uso legítimo de Turbo Pascal requiere la correspondiente licencia.

- Un *compilador* del lenguaje Turbo Pascal que cumple, salvo pequeñas excepciones, la sintaxis y semántica de Pascal estándar. Existe la posibilidad de compilar en memoria o en disco. La primera opción permite alcanzar una gran velocidad de compilación, mientras que la segunda se utiliza para crear los programas ejecutables.

- Un *depurador* que permite realizar un seguimiento de los programas, ejecutándolos paso a paso, deteniendo la ejecución del programa e inspeccionando sus objetos.

- Una *ayuda* a la que se puede acceder desde el entorno, que permite la consulta rápida de la sintaxis y semántica de Turbo Pascal.

- Desde el entorno se puede acceder al DOS, para realizar tareas propias del sistema operativo, sin tener que abandonar el trabajo en curso.

Este entorno está controlado por menús, es decir, el programador puede elegir en cualquier momento entre una serie de opciones organizadas jerárquicamente. Así, en algunos casos, al escoger una opción se abre un submenú que muestra las nuevas (sub)opciones disponibles.

El entorno está basado en ventanas que pueden estar asociadas a programas (pudiendo trabajar con varios a la vez, transfiriendo información de unos a otros), mensajes u operaciones.

Pero para conocerlo, lo mejor es practicar, y eso mismo es lo que proponemos en el apartado siguiente.

C.2 Desarrollo completo de un programa en Turbo Pascal

En este apartado vamos a describir la forma adecuada y eficiente para escribir, almacenar y modificar un programa, para compilarlo y ejecutarlo y para depurarlo.

C.2.1 Arranque del entorno

En primer lugar tenemos que arrancar el entorno, para ello pasamos al directorio donde se encuentre, por ejemplo `PASCAL`. En la versión 7.0, el compilador se encuentra dentro del directorio `\BIN` que a su vez está dentro del directorio `\PASCAL`. Hacemos:

```
C:\> CD PASCAL ↵
```

Figura C.1.

o

```
C:\> CD PASCAL\BIN ↵
```

para la versión 7.0

A continuación arrancamos el entorno tecleando `TURBO`:

```
C:\PASCAL\BIN> TURBO ↵
```

Aparece la pantalla inicial, mostrada en la figura C.1. La línea superior es el menú, es decir, el conjunto de opciones que se pueden ejecutar. En el centro aparece la ventana de edición, con un nombre de archivo por defecto, y debajo una línea que muestra los atajos disponibles, o sea, aquellas teclas que nos permiten efectuar ciertas acciones con una o dos pulsaciones.

Para acceder a las opciones del menú se pulsa [F10] y a continuación su inicial (o la letra resaltada en su caso). Para salir de la barra de menús y editar (crear o modificar) nuestro programa pulsamos [Esc], entonces el cursor pasa a la parte interior de la ventana de edición, que es donde vamos a escribir nuestros programas. Todas las operaciones pueden realizarse igualmente utilizando el ratón.

Los números de la esquina inferior izquierda expresan la fila y columna en que se encuentra el cursor. El número de la esquina superior derecha expresa la ventana que está activa. Turbo Pascal puede tener varios programas abiertos

a la vez en distintas ventanas. También se utilizan ventanas para realizar el seguimiento y depuración de nuestros programas, y para el envío de mensajes.

Inicialmente Turbo Pascal asigna un nombre al fichero de trabajo, que para la ventana 1 es `NONAME00.PAS`

C.2.2 Edición del programa fuente

Se realiza escribiendo las sucesivas líneas del programa, terminando cada línea pulsando la tecla ↵. El alineamiento se realiza la primera vez con la tecla de tabulación y a partir de entonces se hace automáticamente en cada salto de línea.

Para corregir o modificar texto se utilizan las teclas habituales:

teclas de cursor	Para moverse por la ventana.
[DEL] o [SUPR]	Borra la letra que tiene el cursor debajo.
tecla de Retroceso	Borra retrocediendo hacia la izquierda.
[INSERT]	Elige el modo de inserción
[F10]	Sale de la ventana de edición.

Si se elige el modo de inserción, lo que escribamos va desplazando el texto escrito a la derecha desde la posición de inserción (si es que lo hay). En el modo de no inserción, el texto que escribamos ocupa el lugar del texto escrito, por lo que éste último se pierde.

Utilizando las teclas citadas anteriormente hemos escrito el programa para el cálculo recursivo del factorial (véase el apartado 10.1), que aparece en la figura C.2.

Dentro del menú principal existe una opción EDIT (Editar) que ofrece algunas posibilidades complementarias del editor que permiten copiar y pegar fragmentos de texto. Sus opciones trabajan sobre bloques de texto que hay que marcar pulsando la tecla de mayúsculas y las del cursor.

Existe un almacenamiento temporal de texto llamado el portapapeles (en inglés *clipboard*) donde se guardan los bloques de texto cortados (CUT) o copiados (COPY) hasta que se peguen (PASTE) en otro sitio o se corte o copie un nuevo bloque.

C.2.3 Grabar el programa fuente y seguir editando

Para no perder el trabajo que se va realizando, es necesario ir grabando el texto escrito periódicamente. Para ello se utiliza la opción SAVE del menú EDIT, o bien la tecla [F2]. La primera vez que se hace esto, Turbo Pascal

Figura C.2.

muestra la ventana de la figura C.3 (aunque con otra lista de archivos, con toda probabilidad) en la que se nos invita a asignar un nombre al programa. Para movernos por las distintas opciones de la ventana usamos la tecla [Tabulador], salimos con [Esc] y elegimos con la tecla ↵.

Supongamos que queremos llamar a nuestro archivo `FACT` y queremos grabarlo en el directorio raíz de la unidad `A`. Entramos en el apartado Save file as y escribimos el nombre que queremos darle:

```
A:\FACT
```

No es necesario darle extensión, ya que por defecto se le asigna `.PAS`. Si no ponemos unidad y directorio, Turbo Pascal toma los que tiene asignados como directorio de trabajo. Al pulsar la tecla ↵ se graba el programa.

A partir de este momento, cada vez que queramos actualizar en el disco el archivo con el programa, usando el mismo nombre, bastará con pulsar [F2]. Se recomienda actualizar el programa cada diez o veinte minutos y, en cualquier caso, siempre antes de compilar, para evitar la pérdida accidental de parte de nuestro trabajo. Al hacerlo, se graba una copia del programa primitivo con la extensión `.BAK` y de la nueva versión con la extensión `.PAS`.

La tecla [F2] es un atajo de la opción File (Archivo) del menú principal donde se encuentran aquellas opciones necesarias para crear, almacenar e imprimir archivos, salir al DOS y terminar una sesión de trabajo. Por su interés las hemos resumido a continuación:

Figura C.3.

NEW	Abre una nueva ventana de trabajo.
OPEN... (F3)	Lee un archivo.
SAVE (F2)	Almacena el texto en un archivo con el nombre actual.
SAVE AS...	Almacena el texto en un archivo con un nuevo nombre.
SAVE ALL	Almacena en archivos todos los textos del entorno.
CHANGE DIR...	Cambia el directorio de trabajo.
PRINT	Imprime el programa activo.
PRINTER SETUP...	Configura la salida para diferentes impresoras.
DOS SHELL	Sale al DOS (se vuelve al entorno escribiendo EXIT).
EXIT (ALT+X)	Finaliza la ejecución de Turbo Pascal.

La diferencia entre las opciones NEW, OPEN y SAVE es que la primera sirve para abrir una nueva ventana de edición, la segunda para abrir una ventana con un archivo que fue creado con anterioridad, y que es leído desde el disco, y la tercera para guardar un archivo. Para cambiar el nombre del programa con el que estamos trabajando, por ejemplo, cuando se van a introducir cambios que no se sabe si serán definitivos, usamos SAVE AS.

C.2.4 Compilación

En esta fase del desarrollo del programa vamos a realizar la traducción de nuestro programa fuente en Pascal (usualmente almacenado en un archivo con extensión `PAS`) a un programa objeto, ejecutable, que al compilarse en el disco tendrá la extensión `EXE` (véase el apartado 5.3 de [PAO94]).

Activamos el menú Compile haciendo:

`[Alt] + [C]` o bien `[F10] [C]` y después pulsamos `[C]`

o, de otro modo, más cómodamente

`[Alt] + [F9]`

Si el programa se compila con éxito, aparece el mensaje:

```
Compile successful:  Press any key
```

si no, habrá que corregir los errores que vayan apareciendo. Éstos son mostrados por el compilador mediante mensajes de error en la ventana de edición.

Veamos un ejemplo: si por un descuido olvidamos declarar la variable global `n`, al intentar leerla, se produce un error:

```
Error 3:  Unknown identifier.
```
(identificador desconocido)

situándose el cursor en la línea en que se ha detectado el error, debajo de la instrucción `ReadLn(n)`.

El compilador no siempre es capaz de señalar la posición del error con precisión, por lo que en ciertos casos hay que indagar su origen por encima de donde se señala.

El menú Compile consta de varias opciones interesantes que resumimos a continuación:

Compile (Alt+F9)	Compila el programa fuente en curso
Make (F9)	Compila los archivos de programa modificados
Build	Compila todos los archivos de programa
Destination Memory	Permite elegir destino, memoria o disco
Primary file...	Define el programa primario para Make y Build
Clear primary file	Borra el programa primario
Information...	Muestra información sobre la compilación en curso

Con Make y Build se compila el archivo de programa y en su caso aquellos otros archivos de programa que dependan de él o a los que haga referencia, como, por ejemplo, las unidades definidas por el programador (véase el apartado B.11.2).

Durante la depuración, la compilación se puede efectuar almacenando el programa ejecutable en memoria RAM, lo que permite una mayor velocidad.

Una vez que el programa esté totalmente depurado puede compilarse en disco, creándose el archivo ejecutable. Para esto hay que cambiar el destino de la compilación de la siguiente forma: abrimos el menú COMPILE y pulsamos [D] activándose la orden DESTINATION que tiene dos opciones, MEMORY y DISK. Seleccionando DISK las ulteriores compilaciones se dirigirán siempre a disco.

El programa objeto ejecutable tendrá el mismo nombre que el archivo en el que se encuentre el programa fuente, pero con la extensión `.EXE`.

Una vez que hemos compilado (en memoria o en disco) el programa con éxito ya se puede ejecutar.

C.2.5 Ejecución

Para ejecutar el programa se activa el menú RUN con [ALT] + [R] o [F10] [R] y se selecciona la orden RUN volviendo a pulsar [R], o directamente con [CTRL] + [F9].

Desde el entorno integrado, al terminar el programa se vuelve a la ventana de edición sin tiempo para ver las salidas. Podemos ver la ventana de salida tecleando [ALT] + [F5].

Veamos un ejemplo de ejecución del programa `Fact`:

```
Turbo Pascal Version 7.0 Copyright (c) 1983,92 Borland International
Escriba un número natural pequeño:  5
El factorial de 5 es 120
```

C.2.6 Depuración

Durante el proceso de compilación y ejecución de un programa es frecuente que se originen errores que no sean fáciles de corregir. Para ayudar al programador en este caso, el depurador integrado de Turbo Pascal permite analizar el funcionamiento de nuestro programa, ejecutarlo paso a paso, examinar y modificar variables y fijar puntos de ruptura (en los que se detiene la ejecución del programa de forma condicional o incondicional, permitiendo inspeccionar el estado del mismo).

En primer lugar, tenemos que activar el depurador desde el menú de opciones haciendo [F10] [OPTIONS] [DEBUGGER] y marcando la opción INTEGRATED.

Para que el compilador genere la información necesaria para el depurador, hemos de asegurarnos de que la opción DEBUG INFORMATION está marcada, y si tenemos objetos locales, comprobar también la opción LOCAL SYMBOLS dentro de las opciones de la pantalla de opciones del depurador: [F10] [OPTIONS] [COMPILER] [DEBUGGING] (véase el apartado C.3.3).

Figura C.4.

A continuación recompilamos el programa y, al ejecutarlo, el depurador asume el control del programa.

Para ejecutar el programa paso a paso, sin entrar en las llamadas a la función, elegimos la opción [F10] [Run] [Step over] o simplemente pulsamos repetidamente [F8], viendo cómo las líneas del programa se van iluminando y ejecutando sucesivamente. La pantalla alterna entre la ventana de edición y la de salida. Podemos ver que las llamadas recursivas a la función se resuelven en un solo paso.

Para ejecutar paso a paso, pero entrando en las llamadas a subprogramas, elegimos la opción [F10] [Run] [Trace into] o pulsamos repetidamente [F7], viendo cómo las líneas de la función se iluminan al irse ejecutando. En el ejemplo del factorial, la función se repite varias veces debido a las llamadas recursivas. Hay que tener cuidado y no pasarse del final del programa, ya que en tal caso se vuelve a comenzar.

La ejecución paso a paso tiene su complemento perfecto con la inspección de constantes, variables y parámetros, que nos permitirá examinar los valores que tienen estos objetos tras la ejecución de cada instrucción. Para ello hay que abrir una ventana de inspección llamada Watches, en la que, en nuestro ejemplo, colocaremos los identificadores `n` y `num`. Hacemos [F10] [Debug] [Watch], con lo que se abre la ventana, y después [F10] [Debug] [Add Watch] o simplemente [Ctrl]+[F7], lo que permite introducir los identificadores que se deseen.

Si se ejecuta el programa paso a paso, se verán los valores adoptados en cada momento por los objetos incluidos en la ventana de inspección. En el ejemplo, al comenzar el programa tanto `n` como `num` son etiquetados como identificadores desconocidos (`Unknown identifier`), para cambiar posteriormente, adoptando `n` el valor introducido por el usuario, y reduciéndose `num` en cada llamada recursiva.

En la figura C.4 vemos la apariencia de la ventana Watches en un punto intermedio del proceso, donde las sucesivas llamadas recursivas han ido reduciendo el valor del argumento de la función hasta alcanzar el caso base.

Figura C.5.

Una opción interesante en casos como éste es la de CALL STACK (llamada a la pila) que permite ver la sucesión de llamadas realizadas. Recordemos que los objetos locales de las llamadas a subprogramas son almacenados en una estructura de datos del tipo pila (véase el apartado 17.2.3). Esta opción se activa haciendo [F10] [DEBUG] [CALL STACK].

En la figura C.5 se muestra la serie de llamadas producidas para calcular el factorial de 4, empezando por el propio programa, dentro de la ventana CALL STACK.

El depurador permite también detener la ejecución de un programa para la inspección de objetos sin necesidad de ejecutar paso a paso. Esto se hace estableciendo un punto de ruptura BREAKPOINT.

En el ejemplo, fijaremos dicho punto al final de la función, de forma condicional para un valor de `num = 0`. De esta forma se efectuarán las sucesivas llamadas a la función, deteniéndose el programa cuando la condición se haga verdadera.

Haciendo [F10] [DEBUG] [BREAKPOINTS], aparece la ventana de edición de los puntos de ruptura en la que, eligiendo la opción [EDIT], podremos escribir el número de línea y la condición. A continuación salimos de la ventana y ejecutamos el programa. Éste se detiene en el punto de ruptura y muestra un mensaje, lo que nos permite inspeccionar los valores de sus variables y parámetros, así como las llamadas existentes en la pila.

C.2.7 Salida de Turbo Pascal

Para terminar una sesión de trabajo con el entorno integrado de desarrollo, se teclea [F10] [FILE] [EXIT]. Es importante no olvidarse de actualizar el programa en el disco si hemos hecho cambios.

C.3 Otros menús y opciones

En este apartado se hace un breve resumen de otros menús y opciones interesantes de Turbo Pascal que no se han expuesto en el apartado anterior.

La explicación completa de todos los menús y opciones de Turbo Pascal rebasa los objetivos de este apéndice, por lo que debe acudirse a la bibliografía complementaria.

C.3.1 Search (Búsqueda)

Las opciones correspondientes a este menú se utilizan para buscar y sustituir caracteres, palabras o frases, y para buscar líneas, errores y procedimentos o funciones. Son un complemento del propio editor de texto de Turbo Pascal.

C.3.2 Tools (Herramientas)

El menú Tools permite la utilización simultánea de Turbo Pascal y de otros programas complementarios no integrados en el mismo. Estos otros programas pueden servir, por ejemplo, para realizar búsquedas de texto, programar en lenguaje ensamblador, usar un programa de depuración más potente o para analizar y optimizar el funcionamiento de los programas.

C.3.3 Options (Opciones)

El menú Options es uno de los más importantes para el correcto funcionamiento del entorno, pues permite configurar muchos de sus parámetros, que de ser incorrectos dan lugar a errores que impiden la compilación.

Las opciones se seleccionan con la tecla de tabulación y se activan o desactivan pulsando la tecla de espacio (aparece una cruz o quedan en blanco).

Dentro del menú de opciones nos interesan especialmente algunas de las correspondientes al compilador (incluidas en el submenú Compiler), que exponemos a continuación:

- [ ] Force far calls

  Permite llamadas fuera del segmento actual de instrucciones. Debe marcarse al usar procedimientos y funciones como parámetros (véase el apartado A.1).

- [ ] Range checking

  Comprueba si los índices de arrays y cadenas se encuentran dentro de sus límites, y si las asignaciones a variables de tipo escalar no estan fuera de

sus intervalos declarados. Debe activarse cuando puedan aparecer errores de este tipo.

- [ ] OVERFLOW CHECKING

  Comprueba errores de desbordamiento después de efectuar las siguientes operaciones: `+, -, *, Abs, Sqr, Succ` y `Pred`. Debe activarse cuando puedan aparecer errores de este tipo.

- [ ] COMPLETE BOOLEAN EVAL

  Realiza la evaluación completa de las expresiones booleanas sin optimizarlas, es decir, la evaluación del circuito largo (véase el apartado 3.5) en caso de estar activada y del circuito corto en caso contrario.

- [ ] DEBUG INFORMATION

  Genera información de depuración imprescindible para ejecutar el programa paso a paso y para fijar puntos de ruptura. Debe activarse para depurar.

- [ ] LOCAL SYMBOLS

  Genera información de los identificadores locales necesaria para examinar y modificar las variables locales de un subprograma y para ver las llamadas producidas hasta llegar a un determinado subprograma con la opción DEBUG CALL STACK. Debe activarse para depurar dentro de un subprograma.

- [ ] 8087/80287

  Genera código para el coprocesador numérico y debe estar activada para poder utilizar los tipos reales de Turbo Pascal.

- MEMORY SIZES

  Fija los tamaños de memoria de la pila y el montículo, que son dos estructuras necesarias para el funcionamiento de los programas. El tamaño de la pila (STACK SIZE) puede ser insuficiente en programas (normalmente recursivos) con gran numéro de llamadas a subprogramas, como la función de Ackermann (véase el apartado 10.3.3). En dichos casos puede aumentarse su tamaño hasta un valor máximo de 65535.

- DEBUGGER

  En esta opción se fijan ciertos parámetros del depurador, entre otros, si se usa el depurador integrado o independiente.

- Directories

  Aquí se fijan los directorios de trabajo de los diferentes archivos que se utilizan en Turbo Pascal. Para situar más de un directorio en la lista, se separan mediante punto y coma.

- Environment

  Abre un submenú donde se pueden fijar muchos de los parámetros de funcionamiento del entorno integrado, parámetros de pantalla, del editor, del ratón, al arrancar, colores, etc. Esta configuración se guarda en el archivo `TURBO.TP`

C.3.4 Window (Ventana)

El menú Window abre un submenú para el control de las ventanas del entorno integrado. Se puede elegir su disposición, modificar el tamaño y posición y elegir la ventana activa, entre otras opciones.

C.3.5 Help (Ayuda)

El entorno integrado de Turbo Pascal dispone de una extensa ayuda integrada que incluye la explicación de todos sus mandatos, los del lenguaje Pascal (con las extensiones de Turbo Pascal) y otros aspectos como unidades y directivas de compilación.

El texto de la ayuda tiene formato de *Hipertexto*, es decir, ciertas palabras aparecen resaltadas, y basta con situar el cursor sobre ellas y pulsar la tecla ↵ para acceder a información concreta sobre el concepto indicado.

La ayuda de Turbo Pascal es dependiente del contexto, o sea, se abre en el apartado correspondiente al mandato con el que estamos trabajando o en la palabra señalada por el cursor. No obstante, si queremos buscar otros temas, podemos acudir al menú de ayuda.

Además, la ayuda dispone de información sobre los mensajes de error, sobre el uso (sintaxis y semántica) de los identificadores predefinidos, palabras reservadas, etc., constituyendo un verdadero manual del usuario.

C.4 Ejercicios

1. Invitamos al lector a que utilice el entorno de Turbo Pascal para desarrollar gradualmente los programas que se han propuesto como ejercicios en los sucesivos capítulos del libro.

2. Utilice los tipos numéricos enteros de Turbo Pascal para el cálculo del factorial. Determine cuál es el número mayor del que se puede calcular el factorial, en los tipos `byte`, `integer`, `word`, `shortint` y `longint`.
3. Calcule el valor de π y del número e, con la máxima precisión posible utilizando el tipo real `extended` y la suma de las series que se presentan a continuación:
   $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} + \ldots$
   $e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$
4. Utilice las cadenas de caracteres para comprobar si una frase dada forma un palíndromo.
5. Escriba una procedimiento que haga más robusta la entrada de valores numéricos enteros utilizando el procedimiento `Val`. Para ello el procedimiento leerá el número como una cadena de caracteres. Si se produce algún error en la introducción deberá mostrar la parte correcta y pedir el resto del número.
6. Compile los programas ejemplo con paso de subprogramas como parámetros (veáse el apartado A.3) en Turbo Pascal.
7. Escriba un programa en Turbo Pascal que pida la unidad, directorio, nombre y extensión de un archivo de texto, lo abra para escritura y lea repetidamente una cadena desde teclado y la escriba en dicho archivo, añadiendo un salto de línea al final de la misma, finalizando la introducción cuando se escriba una cadena vacía.
8. Escriba un programa en Turbo Pascal que pida la unidad, directorio, nombre y extensión de un archivo de texto existente y muestre su contenido en pantalla, incluyendo los saltos de línea. Asimismo, al terminar la lectura del archivo, el programa mostrará un resumen indicando el total de caracteres leídos, cuántos son respectivamente mayúsculas, minúsculas, números u otros símbolos, y cuántos saltos de línea tenía.

C.5 Referencias bibliográficas

La explicación más directa, sobre el funcionamiento del entorno y del lenguaje Turbo Pascal la podemos encontrar en su propia ayuda interactiva, donde acudiremos por su inmediatez para solventar aquellas dudas que pueden aparecer mientras utilizamos el entorno. No obstante, no es recomendable utilizar la ayuda como texto para el aprendizaje de Turbo Pascal, siendo conveniente acudir a otras obras estructuradas de una forma más pedagógica.

Para utilizar un compilador tan completo y extenso como Turbo Pascal no hay nada mejor que disponer de los manuales originales [Bor92b], [Bor92d], [Bor92a] y [Bor92c].

El libro [ON93] está concebido como un completo manual de referencia de Turbo Pascal cubriendo todos los aspectos del entorno, con ejemplos completos.

El texto [Joy90] es un manual de programación en Turbo Pascal que ofrece una visión completa del mismo en sus versiones 4.0, 5.0 y 5.5.

El libro [CGL+94] esta orientado a un primer curso de programación estructurada y orientada a objetos utilizando como lenguaje el Turbo Pascal y cuenta con numerosos ejercicios.

Bibliografía

[AA78] S. Alagíc y M.A. Arbib. *The design of well-structured and correct programs*. Springer Verlag, 1978.

[AHU88] A. V. Aho, J. E. Hopcroft, y J. Ullman. *Estructuras de datos y algoritmos*. Addison-Wesley Iberoamericana, 1988.

[AM88] F. Alonso Amo y A. Morales Lozano. *Técnicas de programación*. Paraninfo, 1988.

[Arn94] D. Arnow. Teaching programming to liberal arts students: using loop invariants. *SIGCSE Bulletin of the ACM*, 3:141–144, 1994.

[Bar87] J. G. P. Barnes. *Programación en Ada*. Díaz de Santos, 1987.

[BB90] G. Brassard y P. Bratley. *Algorítmica (concepción y análisis)*. Masson, 1990.

[BB97] G. Brassard y P. Bratley. *Fundamentos de algoritmia*. Prentice-Hall, 1997.

[Ben86] J. Bentley. *Programming pearls*. Addison-Wesley Publishing Company, 1986.

[Bie93] M. J. Biernat. Teaching tools for data structures and algorithms. *SIGCSE Bulletin of the ACM*, 25(4):9–11, Dic. 1993.

[BKR91] L. Banachowski, A. Kreczmar, y W. Rytter. *Analysis of algorithms and data structures*. Addison-Wesley, 1991.

[Bor92a] Borland International Inc. *Turbo Pascal 7.0 Library Reference*, 1992.

[Bor92b] Borland International Inc. *Turbo Pascal 7.0 Programmer's Guide*, 1992.

[Bor92c] Borland International Inc. *Turbo Pascal 7.0 TurboVision Guide*, 1992.

[Bor92d] Borland International Inc. *Turbo Pascal 7.0 User's Guide*, 1992.

[BR75] J. R. Bitner y M. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18(11):651–655, Nov. 1975.

[BW89] R. Bird y P. Wadler. *Introduction to functional programming.* Prentice Hall International, 1989.

[CCM+93] J. Castro, F. Cucker, F. Messeguer, A. Rubio, Ll. Solano, y B. Valles. *Curso de programación.* McGraw-Hill, 1993.

[CGL+94] J. M. Cueva Lovelle, M. P. A. García Fuente, B. López Pérez, M. C. Luengo Díez, y M. Alonso Requejo. *Introducción a la programación estructurada y orientada a objetos con Pascal.* Editado por los autores, 1994.

[CK76] L. Chang y J. F. Korsh. Canonical coin changing and greedy solutions. *Journal of the ACM*, 23(3):418–422, Jul. 1976.

[CM] W. Collins y T. McMillan. Implementing abstract data types in Turbo Pascal.

[CMM87] M. Collado, R. Morales, y J. J. Moreno. *Estructuras de datos. Realización en Pascal.* Díaz de Santos, S. A., 1987.

[Col88] W.J. Collins. The trouble with for-loop invariants. *SIGCSE Bulletin of the ACM*, pages 1–4, 1988.

[Dan89] R. L. Danilowicz. Demostrating the dangers of pseudo-random numbers. *SIGCSE bulletin of the ACM*, 21(2):46–48, Jun. 1989.

[dat73] Datamation. Número especial dedicado a la programación estructurada, Dic. 1973.

[DCG+89] P. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. B. Tucker, A. J. Turner, y P. R. Young. Computing as a discipline. *Communications of the ACM*, 32(1):9–23, 1989.

[DDH72] O. J. Dahl, E. W. Dijkstra, y C. A. R. Hoare. *Structured Programming.* Academic Press Ltd., 1972.

[Dew85a] A. K. Dewney. Cinco piezas sencillas para un bucle y generador de números aleatorios. *Investigación y Ciencia*, 105:94–99, Jun. 1985.

[Dew85b] A. K. Dewney. Ying y yang: recurrencia o iteración, la torre de hanoi y las argollas chinas. *Investigación y Ciencia*, 100:102–107, En. 1985.

[Dij68] E.W. Dijkstra. Goto statement considered harmful. *Communications of the ACM*, 11(3), Mar. 1968.

[DL89] N. Dale y S. Lilly. *Pascal y estructuras de datos*. McGraw-Hill, 1989.

[DM84] B. P. Demidovich y I. A. Maron. *Cálculo Numérico Fundamental*. Paraninfo, Madrid, 1984.

[DW89] N. Dale y C. Weems. *Pascal*. McGraw-Hill, 1989.

[ES85] G. G. Early y D. F. Stanat. Chinese rings and recursion. *SIGCSE Bulletin of the ACM*, 17(4), 1985.

[For82] G. Ford. A framework for teaching recursion. *SIGCSE Bulletin of the ACM*, 14(2):32–39, July 1982.

[Fru84] D. Frutos. Tecnología de la programación. Apuntes de curso. Manuscrito, 1984.

[FS87] G. Fernández y F. Sáez. *Fundamentos de Informática*. Alianza Editorial. Madrid, 1987.

[GGSV93] J. Galve, J. C. González, A. Sánchez, y J. A. Velázquez. *Algorítmica. Diseño y análisis de algoritmos funcionales e imperativos*. Ra-ma, 1993.

[GL86] L. Goldschlager y A. Lister. *Introducción moderna a la Ciencia de la Computación con un enfoque algorítmico*. Prentice-Hall hispanoamericana. S.A. Méjico, 1986.

[GT86] N. E. Gibbs y A. B. Tucker. A model curriculum for a liberal arts degree in computer science. *Communications of the ACM*, 29(3), march 1986.

[Har89] R. Harrison. *Abstract data types in Modula-2*. John Wiley and sons, 1989.

[Hay84] B. Hayes. Altibajos de los números pedrisco. A la búsqueda del algoritmo general. *Investigación y Ciencia*, 90:110–115, Mar. 1984.

[Hig93] T.F. Higginbotham. The integer square root of n via a binary search. *SIGCSE Bulletin of the ACM*, 23(4), Dic. 1993.

[HS90] E. Horowitz y S. Sahni. *Fundamentals of data structures in Pascal*. Computer Science Press, 3 edición, 1990.

[Joy90] L. Joyanes Aguilar. *Programación en Turbo Pascal Versiones 4.0, 5.0 y 5.5.* Mc Graw-Hill, 1990.

[JW85] K. Jensen y N. Wirth. *Pascal user manual and report. Revised for the ISO.* Springer Verlag, 1985.

[KR86] B.W. Kernighan y D.M. Ritchie. *El lenguaje de programación C.* Prentice-Hall, 1986.

[KSW85] E. B. Koffman, D. Stemple, y C. E. Wardle. Recomended curriculum for cs2, 1984. *Communications of the ACM*, 28(8), aug. 1985.

[LG86] B. Liskov y J. Guttag. *Abstraction and interpretation in program development.* MIT Press, 1986.

[Mar86] J. J. Martin. *Data types and structures.* Prentice Hall, 1986.

[McC73] D. D. McCracken. Revolution in programming: an overview. *Datamation*, Dic. 1973.

[Mor84] B. J. T. Morgan. *Elements of simulation.* Chapman and Hall, 1984.

[MSPF95] Ó. Martín-Sánchez y C. Pareja-Flores. A gentle introduction to algorithm complexity for CS1 with nine variations on a theme by Fibonacci. *SIGCSE bulletin of the ACM*, 27(2):49–56, Jun. 1995.

[ON93] S. K. O'Brien y S. Nameroff. *Turbo Pascal 7, Manual de referencia.* Osborne Mc Graw-Hill, 1993.

[PAO94] C. Pareja, A. Andeyro, y M. Ojeda. *Introducción a la Informática (I). Aspectos generales.* Editorial Complutense. Madrid, 1994.

[PJ88] M. Page-Jones. *The practical guide to structured design.* Prentice-Hall, 1988.

[PM88] S. K. Park y K. W. Miller. Random number generators: good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, Oct. 1988.

[Pn93] R. Peña. *Diseño de programas. Formalismo y abstracción.* Prentice Hall, 1993.

[Pre93] R. S. Pressman. *Ingeniería del Software. Un enfoque práctico.* McGraw-Hill, 1993.

[PV87] L. Pardo y T. Valdés. *Simulación. Aplicaciones prácticas en la empresa.* Díaz de Santos, S. A., 1987.

[RN88] L. Råde y R. D. Nelson. *Adventures with your computer*. Penguin Books Ltd., Middlesex, Gran Bretaña, 1988.

[Sal93] W. I. Salmon. *Introducción a la computación con Turbo Pascal*. Addison-Wesley Iberoamericana, 1993.

[Sed88] R. Sedgewick. *Algorithms*. Addison-Wesley, 1988.

[Str84] B. Stroupstrup. *The C++ programming language*. Addison Wesley, 1984.

[Tam92] W. C. Tam. Teaching loop invariants to beginners by examples. *SIGCSE Bulletin of the ACM*, pages 92–96, 1992.

[Tur91] A. J. Turner. Computing curricula (a summary of the ACM/IEEE-CS joint curriculum task force report). *Communications of the ACM*, 34(6):69–84, 1991.

[War92] J. S. Warford. Good pedagogical random number generators. *Communications of the ACM*, pages 142–146, 1992.

[Wei95] M. A. Weiss. *Estructuras de datos y algoritmos*. Addison-Wesley Iberoamericana, 1995.

[Wie88] S. Wiedenbeck. Learning recursion as a concept and as a programming technique. *Communications of the ACM*, 1988.

[Wil89] H. S. Wilf. *Algorithmes et complexité*. Masson, 1989.

[Wir86] N. Wirth. *Algoritmos + Estructuras de datos = Programas*. Ediciones del Castillo. Madrid, 1986.

[Wir93] N. Wirth. Recollections about the development of Pascal. *ACM Sigplan Notices*, 28(3):1–19, 1993.

[Wri75] J. W. Wright. The change making problem. *Journal of the ACM*, 22(1):125–128, Jan. 1975.

[Yak77] S. J. Yakowitz. *Computational probability and simulation*. Addison-Wesley Publishing Company, 1977.

Índice alfabético