

NIKLAUS WIRTH

**ALGORITMOS
+ ESTRUCTURAS DE DATOS
= PROGRAMAS**

TABLA DE MATERIAS

PRESENTACION INTRODUCCION DE LOS TRADUCTORES PROLOGO

1.	ESTRUCTURAS FUNDAMENTALES DE DATOS	1
1.1.	Introducción	1
1.2.	Concepto de tipo de datos	4
1.3.	Tipos elementales de datos	7
1.4.	Tipos elementales normalizados	8
1.5.	Tipos subcampo	11
1.6.	La estructura array	12
1.7.	La estructura registro	17
1.8.	Variantes de las estructuras registro	22
1.9.	La estructura conjunto (set)	25
1.10.	Representación de las estructuras array, registro y conjunto	30
1.10.1.	<i>Representación de los arrays</i>	32
1.10.2.	<i>Representación de las estructuras registro</i>	35
1.10.3.	<i>Representación de los conjuntos</i>	36
1.11.	La estructura fichero secuencial	37
1.11.1.	<i>Operadores elementales de ficheros</i>	40
1.11.2.	<i>Ficheros con subestructura</i>	43
1.11.3.	<i>Textos</i>	45
1.11.4.	<i>Un programa para edición de ficheros</i>	53
	Ejercicios	57
	Referencias	59
2.	ORDENACION	61
2.1.	Introducción	61
2.2.	Ordenación de arrays	64
2.2.1.	<i>Ordenación por inserción directa</i>	65
2.2.2.	<i>Ordenación por selección directa</i>	68
2.2.3.	<i>Ordenación por intercambio directo</i>	71
2.2.4.	<i>Ordenación por inserción con incrementos decrecientes</i>	74
2.2.5.	<i>Ordenación según un árbol</i>	76
2.2.6.	<i>Ordenación por partición</i>	82
2.2.7.	<i>Obtención de la mediana</i>	89
2.2.8.	<i>Comparación de los métodos de ordenación de arrays</i>	91
2.3.	Ordenación de ficheros secuenciales	93
2.3.1.	<i>Método de mezcla directa</i>	93
2.3.2.	<i>Mezcla natural</i>	99
2.3.3.	<i>Mezcla equilibrada múltiple</i>	106
2.3.4.	<i>Ordenación polifásica</i>	112
2.3.5.	<i>Distribución de los tramos iniciales</i>	125
	Ejercicios	130
	Referencias	132

3. ALGORITMOS RECURSIVOS

3.1. Introducción	134
3.2. Cuándo no utilizar la recursión	137
3.3. Dos ejemplos de programas recursivos	140
3.4. Algoritmos de «vuelta atrás»	147
3.5. El problema de las ocho reinas	153
3.6. El problema de los matrimonios estables	158
3.7. El problema de la selección óptima	165
Ejercicios	169
Referencias	171

4. ESTRUCTURAS DINAMICAS DE INFORMACION

4.1. Tipos recursivos de datos	173
4.2. Punteros o referencias	177
4.3. Listas lineales	182
4.3.1. Operaciones básicas	182
4.3.2. Listas ordenadas y listas reorganizables	186
4.3.3. Una aplicación: ordenación topológica	194
4.4. Estructuras árbol	202
4.4.1. Definiciones y conceptos básicos	202
4.4.2. Operaciones básicas con árboles binarios	211
4.4.3. Búsqueda e inserción en árboles	215
4.4.4. Borrado en árboles	224
4.4.5. Análisis de la búsqueda e inserción en árboles	226
4.4.6. Árboles equilibrados	229
4.4.7. Inserción en árboles equilibrados	231
4.4.8. Borrado en árboles equilibrados	236
4.4.9. Árboles de búsqueda óptimos	240
4.4.10. Presentación de estructuras árbol	247
4.5. Árboles multicamino	258
4.5.1. Árboles B	261
4.5.2. Árboles B binarios	273
4.6. Transformaciones de claves («hashing»)	281
4.6.1. Elección de la función de transformación	282
4.6.2. Manejo de colisiones	283
4.6.3. Análisis del Método de transformación de claves	288
Ejercicios	291
Referencias	296

5. ESTRUCTURAS Y COMPILADORES DE LENGUAJES

5.1. Definición y estructura de los lenguajes	297
5.2. Análisis de frases	300
5.3. Construcción de un grafo sintáctico	306
5.4. Construcción de un analizador para una sintaxis dada	310
5.5. Construcción de un programa analizador dirigido por tabla	314
5.6. Un traductor de BNF en estructuras de datos para dirigir analizadores	318
5.7. El lenguaje de programación PL/0	326
5.8. Un analizador sintáctico para PL/0	330
5.9. Tratamiento de errores sintácticos	339
5.10. Un procesador PL/0	350
5.11. Generación de código	354
Ejercicios	367
Referencias	369

APENDICES

A EL CONJUNTO DE CARACTERES ASCII	371
B DIAGRAMAS SINTACTICOS DE PASCAL	372
INDICE DE MATERIAS	379
INDICE DE PROGRAMAS	382

PRESENTACION

Debido a la complejidad de los temas y al proceso de cambio constante en el campo de la informática, las publicaciones en este área oscilan habitualmente entre los simples manuales de programación y la monografía excesivamente especializada, no siendo, en cambio, fácil encontrar textos adecuados para la enseñanza de programación que aúnen modernidad, generalidad y profundidad en el tratamiento, con funcionalidad didáctica.

Un texto que cumple las condiciones anteriores, como lo demuestran las siete ediciones en menos de tres años que se han llevado a efecto de su versión inglesa, es el presente libro del profesor Wirth, que incorpora los desarrollos últimos, tanto en lo referente a la gama de algoritmos que describe, como a su metodología de construcción.

La Facultad de Informática, que sigue una línea de apoyo a la creación de una base auténticamente científica de la informática en nuestro país, es consciente de la importancia que tiene este libro tanto para la consulta de los profesionales como para texto de enseñanza de programación en los centros universitarios y, por ello, no ha dudado en impulsar la iniciativa de los profesores Angel Alvarez Rodriguez y José Cuena Bartolomé de esta Facultad de realizar su versión castellana, considerando la indudable calidad de esta publicación.

RAFAEL PORTAENCASA

DECANO DE LA FACULTAD DE INFORMATICA DE MADRID

1 ESTRUCTURAS FUNDAMENTALES DE DATOS

1.1. INTRODUCCION

Los modernos computadores digitales fueron inventados con la idea de facilitar y acelerar cálculos complicados y onerosos de tiempo. En la mayoría de las aplicaciones, su capacidad de almacenamiento y acceso a grandes masas de información juega un papel dominante y, por ello, se considera ésta su característica primordial, es decir, su capacidad para realizar cálculos aritméticos ha llegado a tener escaso relieve en muchos casos.

En todos estos casos, la gran masa de información que es necesario procesar representa, en cierta forma, una *abstracción* de una parte del mundo real. La información utilizable por el computador consiste en una selección de *datos* de la realidad, precisamente el conjunto de datos que se considera relevante para el problema en estudio, y a partir del cual se cree que pueden obtenerse los resultados deseados. Los datos representan una abstracción de la realidad en el sentido de que ciertas propiedades y características de los objetos reales se ignoran porque no interesan para el problema concreto que se estudia. Por ello, una abstracción es, también, una simplificación de la realidad.

Puede tomarse como ejemplo el fichero de personal de una empresa. Cada empleado se representa (en forma abstracta) en este fichero por un conjunto de datos de interés bien para la empresa o para sus procesos contables. Esta información puede incluir datos de identificación del empleado, por ejemplo, su nombre y sueldo. Pero, seguramente, no contendrá datos sin interés tales como el color del pelo, peso y estatura.

Al resolver un problema, se utilice o no el computador, es necesario elegir una abstracción de la realidad, es decir, definir un conjunto de datos para representar la situación real. Esta elección debe estar guiada por el problema a resolver. A continuación, debe seleccionarse la forma de representar esta información. Esta

segunda elección debe hacerse en base al instrumento con que va a resolverse el problema, por ejemplo en base a las posibilidades concretas que ofrece el computador. En muchos casos, estas dos etapas de diseño no son totalmente independientes una de otra.

La elección de la representación de los datos es a menudo bastante difícil y no está determinada exclusivamente por los instrumentos disponibles. Deben tenerse siempre en cuenta las operaciones a realizar con los datos. Un buen ejemplo ilustrativo de esto es la representación de los números que son, a su vez, abstracciones de propiedades de objetos a caracterizar. Si la adición es la única (o al menos predominante) operación a realizar, una buena forma de representar el número n es escribir n barras. La regla de adición, con esta representación, es realmente obvia y muy sencilla. La numeración romana se basa en el mismo principio sencillo y las reglas para sumar son igualmente simples para números pequeños. En cambio, la numeración árabe requiere reglas que están lejos de ser obvias (para números pequeños) y por tanto deben ser aprendidas de memoria. Sin embargo, la situación se invierte cuando se considera la suma de grandes números o la multiplicación y división. La descomposición de estas operaciones en otras más simples es mucho más fácil en el caso de la numeración árabe, debido a su principio sistemático basado en el distinto peso posicional de los dígitos.

Es bien sabido que los computadores usan una representación interna basada en dígitos binarios (bits). Esta representación no es viable para las personas debido al número, frecuentemente elevado, de dígitos que son necesarios pero, en cambio, es muy adecuada para los circuitos electrónicos porque los dos valores 0 y 1 pueden representarse en forma conveniente y fiable por la presencia o ausencia de corrientes eléctricas, carga eléctrica y campos magnéticos.

A partir de este ejemplo, puede verse también que el problema de representación, a menudo, abarca varios niveles de detalle. Dado el problema de representar, por ejemplo, la posición de un objeto, la primera decisión puede conducir a la elección de una pareja de números reales que sean, por ejemplo, sus coordenadas cartesianas o polares. La segunda decisión puede llevar a una representación en coma flotante, donde cada número real se describe por un par de números enteros que representan una fracción f y un exponente e al que se eleva determinada base (por ejemplo, $x = f \cdot 2^e$). La tercera decisión, basada en el conocimiento de que los datos se almacenarán en un computador, puede conducir a una representación binaria posicional de los números enteros y la decisión final podría ser representar los dígitos binarios por la dirección del flujo en un dispositivo magnético de almacenamiento. Evidentemente, la primera decisión en esta cadena está principalmente influida por el planteamiento del problema y las siguientes son cada vez más dependientes de la herramienta a utilizar y su tecnología. Así pues, difícilmente podrá pedirse a un programador que decida sobre la forma de representar los números que deben emplearse o, incluso, sobre las características del dispositivo magnético de almacenamiento. Estas «decisiones de bajo nivel» pueden encomendarse a los diseñadores de equipos que tienen a su disposición la más completa información sobre la tecnología actual, con la que

pueden adoptar una decisión sensata que, en general, será aceptable para todas (o casi todas) las aplicaciones en que intervengan números.

En este contexto adquiere significado el concepto de *lenguajes de programación*. Un lenguaje de programación representa un computador abstracto capaz de entender los términos utilizados en este lenguaje, que pueden ser más abstractos que los de los objetos utilizados por la máquina real. De esta manera, el programador que utiliza este tipo de lenguaje de «alto nivel» estará liberado (e impedido) de tener que considerar cuestiones relativas a la representación de los números, caso de que el número sea un objeto elemental de este lenguaje.

La importancia de utilizar un lenguaje que ofrezca un conjunto conveniente de abstracciones básicas, comunes a la mayoría de los problemas de proceso de datos, reside principalmente en la fiabilidad de los programas resultantes. Es más fácil diseñar un programa razonando sobre los conceptos familiares de números, conjuntos, sucesiones y repeticiones que razonando con bits, «palabras», y saltos de secuencia. Desde luego, un computador real representará todos los datos, ya sean números, conjuntos o sucesiones como grandes masas de bits. Pero esto carece de importancia para el programador, mientras no tenga que preocuparse de los detalles de representación de las abstracciones que ha elegido y pueda estar seguro de que la representación elegida por el computador (o el compilador) es adecuada para sus fines.

Cuanto más próximas son las abstracciones a un computador dado, más fácil le resulta al ingeniero o técnico de implantación del lenguaje la elección de la forma de representación, y más alta es la probabilidad de que una elección única sea adecuada para todas (o casi todas) las aplicaciones posibles. Este hecho determina límites definidos al grado de abstracción a realizar a partir de un computador real dado. Por ejemplo, no tendría sentido incluir objetos geométricos, como elementos básicos de datos, en un lenguaje de uso general, ya que su adecuada representación dependerá, en forma importante, debido a su inherente complejidad, de las operaciones a realizar con tales objetos. La naturaleza y frecuencia de estas operaciones, sin embargo, serán desconocidas para el diseñador de un lenguaje de uso general y de su compilador y, por tanto, cualquier elección que adopte puede no ser apropiada para alguna de las posibles aplicaciones.

Las consideraciones anteriores han determinado en este libro la elección de la notación a utilizar en la descripción de los algoritmos y sus datos. Por un lado, se quiere utilizar nociones familiares en matemáticas, tales como números, conjuntos, sucesiones y similares, más que entidades dependientes del computador tales como secuencias de bits. Igualmente interesa utilizar una notación para la que se conozca la existencia de compiladores eficientes. Tan inadecuado es utilizar lenguajes fuertemente orientados a la máquina y, por tanto, fuertemente dependientes de ella, como describir los programas en una notación abstracta que deje pendientes de resolver los problemas de representación.

El lenguaje de programación PASCAL ha sido diseñado con la intención de encontrar un compromiso entre ambos extremos y se utiliza a lo largo de todo este libro [1.3 y 1.5]. Este lenguaje se ha implantado satisfactoriamente en varios

computadores y se ha comprobado que su notación es lo suficientemente próxima a las máquinas existentes como para que las características elegidas y su representación, puedan explicarse en forma clara. El lenguaje es también suficientemente próximo a otros lenguajes, en particular a ALGOL 60 como para que las lecciones explicadas aquí puedan aplicarse, igualmente bien, para uso de estos lenguajes.

1.2. CONCEPTO DE TIPO DE DATOS

En matemáticas se acostumbra clasificar las variables de acuerdo con ciertas características importantes. Se hacen distinciones claras entre variables lógicas, reales y complejas, o entre variables que representan valores individuales, conjuntos de valores, conjuntos de conjuntos, o entre funciones, funcionales, conjuntos de funciones, y así sucesivamente. Esta noción de clasificación es importante en igual medida, si no más, en proceso de datos.

Se adoptará de aquí en adelante el principio básico de que *cada constante, variable, expresión o función es de un tipo determinado*. Este tipo caracteriza esencialmente el conjunto de valores al que pertenece una constante o sobre el que puede tomar valores una variable o expresión, o cuyos elementos pueden ser generados por una función.

En los textos matemáticos el tipo de una variable se deduce, generalmente de la forma de escritura, sin tener en cuenta el contexto; esto no es factible en los programas, ya que, normalmente, sólo hay un tipo de escritura utilizable en los equipos de proceso de datos (alfabeto latino). Debido a esto, la norma comúnmente aceptada es que el tipo asociado se manifieste explícitamente en una *declaración* de la constante, variable o función y esta declaración preceda en el texto a la utilización de cada constante, variable o función. Esta norma es especialmente oportuna si se tiene en cuenta que un compilador tiene que elegir la representación del objeto en la memoria del computador. Evidentemente, la capacidad de memoria asignada a una variable debe decidirse de acuerdo con el tamaño del campo de valores que la variable puede adoptar. Si esta información es conocida por el compilador, puede evitarse la llamada asignación dinámica de memoria. Esta es muy frecuentemente la clave de una ejecución eficiente de un algoritmo.

Por tanto, las características principales del concepto de tipo que se utiliza a lo largo de este texto, y que están incorporadas en el lenguaje de programación PASCAL, son las siguientes [1.2]:

1. Un tipo de datos determina el conjunto de valores al que pertenece una constante, o que puede tomar una variable o expresión, o que pueden ser generados por un operador o función.
2. El tipo del valor identificado por una constante, variable, o expresión puede deducirse de su forma o de su declaración sin necesidad de ejecutar el proceso.

3. Cada operador o función presupone argumentos de un determinado tipo y produce un resultado también de un tipo determinado. Si un operador admite argumentos de varios tipos (por ejemplo, se utiliza + igualmente para números enteros y reales), el tipo del resultado puede determinarse a partir de reglas específicas del lenguaje.

Por tanto, un compilador puede utilizar esta información sobre tipos para comprobar la compatibilidad y observancia de normas de las distintas construcciones de un programa. Por ejemplo, puede detectarse, sin ejecutar el programa, la asignación de un valor de tipo boolean (lógico) a una variable de tipo aritmético (real). Este tipo de redundancias en el texto de un programa es muy útil en el desarrollo de los mismos y debe considerarse la principal ventaja de los buenos lenguajes de alto nivel sobre los lenguajes en código de máquina (o ensambladores). Evidentemente, los datos acaban representándose por un gran número de dígitos binarios, independientemente de que el programa haya sido concebido inicialmente en un lenguaje de alto nivel, utilizando el concepto de tipo, o en un lenguaje ensamblador que no lo tenga en cuenta. Para el computador, la memoria es una masa homogénea de bits sin estructura aparente. Pero es, precisamente, esta estructura abstracta, la única que permite a los programadores reconocer significado en el monótono paisaje de la memoria de un computador.

La teoría presentada en este libro y el lenguaje de programación PASCAL especifican ciertos métodos de definición de tipos de datos. En la mayoría de los casos se definen nuevos tipos de datos en función de otros definidos previamente. Los valores de tal tipo son usualmente agrupaciones de *valores componentes* de los *tipos constituyentes* definidos previamente y se dice que están *estructurados*. Si solamente hay un tipo constituyente, esto es, si todos los componentes son del mismo tipo, entonces éste se denomina *tipo base*.

El número de valores distintos que pertenecen a un tipo *T* se llama cardinalidad (o cardinal) de *T*. La cardinalidad proporciona una medida de la cantidad de memoria que se necesita para representar una variable *x* de tipo *T*; y esto último se expresa de la forma *x : T*.

Como los tipos constituyentes, a su vez, pueden también estar estructurados, pueden construirse verdaderas jerarquías de estructuras pero, obviamente, los componentes últimos de una estructura deben ser atómicos. Por esto, también es necesaria una notación para introducir tales tipos elementales, no estructurados. Un método directo es el de *enumeración* de los valores que van a constituir el tipo. Por ejemplo, en un programa que trate figuras geométricas planas puede introducirse un tipo elemental llamado *forma* cuyos valores pueden designarse con los identificadores: *rectángulo*, *cuadrado*, *elipse*, *círculo*. Pero además de estos tipos definidos por el programador debe haber algunos *tipos normalizados* (o estándar) que se llaman tipos predefinidos: incluirán frecuentemente los *valores numéricos* y *lógicos*. Si existe un orden entre los valores individuales el tipo se dice que está ordenado o que es un *escalar*. En PASCAL se supone que todos los tipos no estructurados están ordenados; en el caso de enumeración

explicita, los valores se suponen ordenados según la secuencia de enumeración.

Con este instrumento es posible definir tipos elementales y construir agrupaciones, tipos estructurados, hasta el grado de complejidad que se desee. En la práctica no es suficiente tener sólo un método general de combinar los tipos constituyentes en una estructura. Teniendo en cuenta los problemas prácticos de representación y utilización, un lenguaje de programación, de uso general, debe ofrecer varios métodos de *estructuración de datos*. Matemáticamente pueden ser todos equivalentes; se diferenciarán en los operadores utilizados para construir sus valores y para seleccionar componentes de estos valores. Los métodos básicos de estructuración presentados aquí son el *array*^{*}, el *registro*, el *conjunto* y el *fichero* secuencial. No se definen habitualmente estructuras más complicadas como tipos «estáticos»; éstas son, más bien, generadas «dinámicamente» durante la ejecución del programa en la cual pueden variar de tamaño y de forma. Tales estructuras son el tema del capítulo 4 e incluyen listas, anillos, árboles y grafos finitos genéricos.

Las variables y los tipos de datos se introducen en un programa para ser utilizados en proceso. A este fin debe disponerse de un conjunto de *operadores*. Tal como sucede con los tipos de datos, los lenguajes de programación ofrecen un cierto número de operadores elementales normalizados (atómicos) y un número de métodos de estructuración, de forma que pueden definirse operaciones complejas en función de los operadores elementales. La composición de operaciones se considera, a menudo, la tarea fundamental del arte de programar. Sin embargo, en lo sucesivo, se pondrá en evidencia que una apropiada composición de los datos es igualmente fundamental y esencial.

Los operadores básicos más importantes son la *comparación* y la *asignación*, es decir, la comprobación de igualdad (y de orden en el caso de tipos ordenados) y la instrucción que obliga a la igualdad. La diferencia fundamental entre estas dos operaciones se pone de relieve mediante la distinción clara en su notación a lo largo de este texto, aunque, desgraciadamente, esta distinción aparece oscurecida en lenguajes tan ampliamente usados como el FORTRAN y PL/I que utilizan el signo de igualdad como operador de asignación.

Comprobación (test) de igualdad: $x = y$

Asignación a x : $x := y$

Estos operadores fundamentales se definen para la mayoría de los tipos de datos, pero hay que advertir que su ejecución puede involucrar una cantidad importante de esfuerzo de proceso si los datos son extensos y altamente estructurados.

Además del test de igualdad (u orden) y el operador de asignación, existe

* Dada la frecuencia de esta palabra y la dificultad de encontrar un término castellano adecuado para su traducción se adopta la misma palabra inglesa que, por otro lado, es suficientemente familiar a los profesionales de informática. (*N. de los T.*)

una clase de operadores fundamentales, definidos implícitamente, llamados *operadores de transferencia*. Estos aplican unos tipos de datos en otros. Son particularmente importantes en relación con los tipos estructurados. Los valores estructurados se generan a partir de sus valores correspondientes por los llamados *constructores* y los valores componentes son extraídos de los valores estructurados mediante los llamados *selectores*. Constructores y selectores son, así, operadores de transferencia que aplican los tipos constituyentes en los estructurados y recíprocamente. Cada método de estructuración tiene su pareja particular de constructores y selectores que se diferencian claramente por su notación.

Los tipos de datos elementales estandarizados también requieren un conjunto de operadores primitivos normalizados. Así, junto con los tipos de datos normalizados de números y valores lógicos también se introducen las operaciones convencionales de aritmética y lógica proposicional.

1.3. TIPOS ELEMENTALES DE DATOS

En muchos programas los números enteros se utilizan cuando no están involucradas propiedades numéricas y cuando cada entero representa una elección entre un pequeño número de alternativas. Para estos casos, se introduce un nuevo tipo elemental, no estructurado, definido por enumeración del conjunto de todos los posibles valores c_1, c_2, \dots, c_n .

$$\text{type } T = (c_1, c_2, \dots, c_n) \quad (1.1)$$

La cardinalidad de T es $\text{card}(T) = n$.

EJEMPLOS

```

type forma = (rectangulo, cuadrado, elipse, circulo)
type color = (rojo, amarillo, verde)
type sexo = (masculino, femenino)
type boolean = (false, true)
type diasemana = (lunes, martes, miercoles, jueves, viernes, sabado, domingo)
type moneda = (peseta, franco, marco, libra, dolar, lira, rublo, cruzeiro, yen)
type destino = (infierno, purgatorio, cielo)
type vehiculo = (tren, autobus, automovil, barco, avion)
type graduacion = (soldado, cabo, sargento, teniente, capitán, comandante, coronel, general)
type objeto = (constante, tipo, variable, procedimiento, funcion)
type estructura = (fichero, array, registro, conjunto)
type condicion = (manual, descargada, paridad, desalineada)

```

La definición de tales tipos introduce no sólo un nuevo identificador de tipo

sino también el conjunto de identificadores que designan los valores de este nuevo tipo. Estos identificadores pueden ser usados como constantes a lo largo del programa con lo que aumenta de una manera considerable su inteligibilidad. Si, por ejemplo, se introducen las variables *s*, *d*, *g* y *b*.

```
var s: sexo
var d: diasemana
var g: graduacion
var b: boolean
```

es posible escribir las siguientes asignaciones

```
s := masculino
d := domingo
g := comandante
b := true
```

Evidentemente estas instrucciones son mucho más informativas que sus correspondientes en código numérico:

```
s := 1      d := 7      g := 6      b := 2
```

resultantes de suponer que las variables *s*, *d*, *g* y *b* estuvieran definidas como de tipo *integer* y que, a las constantes de los tipos anteriores, se hubiera hecho corresponder el número natural indicativo de su situación en el orden de enumeración. Además, un compilador puede controlar el uso indebido de operadores aritméticos con tales variables, definidas como no numéricas, así como sucede en la instrucción

```
s := s + 1
```

Sin embargo, si se considera un tipo como ordenado, es razonable introducir funciones que generen el predecesor y el sucesor de su argumento. Estas se denominan *pred(x)* y *succ(x)*. El orden de los valores de *T* se define por la regla

$$(c_i < c_j) \equiv (i < j) \quad (1.2)$$

1.4. TIPOS ELEMENTALES NORMALIZADOS

Los tipos elementales normalizados son aquellos que forman parte de la mayoría de los computadores. Comprenden los números enteros, los valores lógicos (verdadero, falso) y un conjunto de caracteres de escritura. En los grandes computadores se incorporan también números fraccionarios y un conjunto ade-

cuado de operadores elementales asociados a ellos. Estos tipos se designan por los identificadores

integer, *boolean*, *char*, *real*

El tipo *integer* comprende un subconjunto de los enteros cuyo tamaño puede variar según el computador de que se trate. Se supone, en cualquier caso, que todas las operaciones, con datos de este tipo, son exactas y se corresponden con las leyes ordinarias de la aritmética, y que el proceso de cálculo se interrumpe cuando aparece un resultado que está fuera del subconjunto de números representado en el computador. Los operadores normalizados son los de las cuatro operaciones básicas de suma (+), resta (-), multiplicación (*) y división (*div*). Esta última se considera que produce un resultado entero, ignorando el resto de la división, de tal forma que para enteros positivos *m* y *n*:

$$m - n < (m \text{ div } n) * n \leq m \quad (1.3)$$

El operador de resto o módulo (*mod*) se define en función de la división de la forma:

$$(m \text{ div } n) * n + (m \text{ mod } n) = m \quad (1.4)$$

De esta forma, *m div n* es el cociente entero de *m* y *n*, y *m mod n* es el resto asociado.

El tipo *real* designa un subconjunto de los números reales. Mientras la aritmética de los números enteros se supone que produce resultados exactos, en la aritmética de tipo *real* se permite una cierta imprecisión, dentro de los límites del error de redondeo producido por el cálculo con un número finito de dígitos. Esta es la principal razón para distinguir explícitamente los tipos *integer* y *real* en la mayoría de los lenguajes de programación.

Para la división de números reales, que produce un cociente real, se emplea el signo de barra oblicua (/) en contraste con el *div* de la división entera.

Los dos valores correspondientes al tipo *boolean* se designan por los identificadores *true* y *false*. Los operadores booleanos son los de las operaciones lógicas de conjunción, unión y negación, cuyos valores se definen en la Tabla 1.1. La con-

<i>p</i>	<i>q</i>	<i>p ∨ q</i>	<i>p ∧ q</i>	$\neg p$
<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>false</i>
<i>true</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>
<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>	<i>true</i>
<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>

Tabla 1.1 Operadores Booleanos.

junción lógica se simboliza por \wedge (o el literal **and**) la unión lógica por \vee (o el literal **or**) y la negación por \neg (o el literal **not**). Obsérvese que las comparaciones son operadores que producen un resultado de tipo *boolean*. Así, el resultado de una comparación puede asignarse a una variable, o puede ser utilizado como operando con un operador lógico en una expresión booleana. Por ejemplo, dadas las variables booleanas p y q y las variables enteras x, y, z , las dos asignaciones:

$$p := x = y$$

$$q := (x < y) \wedge (y \leq z)$$

producen, si los valores de las variables en el momento de proceso, son $x = 5$, $y = 8$, $z = 10$, $p = \text{false}$ y $q = \text{true}$.

El tipo normalizado *char* comprende el conjunto de caracteres imprimibles. Desgraciadamente, no existe ningún conjunto normalizado de caracteres generalmente aceptado para todos los equipos de proceso de datos. Por ello el adjetivo «normalizado» puede ser equívoco en este caso; debe entenderse como normalizado en el equipo de proceso de datos en el que se va a ejecutar determinado programa.

El conjunto de caracteres definido por la International Standards Organization (I.S.O.) y, en particular, su versión americana ASCII (American Standard Code for Information Interchange), es probablemente el código más ampliamente admitido. Por ello este código se presenta en el Apéndice A. Consta de 95 caracteres de impresión (gráficos) y 33 caracteres de control; estos últimos se utilizan principalmente en transmisión de datos y control del equipo de impresión. Un subconjunto de 64 caracteres gráficos (sólo las letras mayúsculas) se utiliza de forma generalizada y se denomina el código *restringido ASCII*.

Para diseñar algoritmos que trabajen con caracteres (es decir, valores de tipo *char*) que sean independientes del equipo de proceso de datos específico, es conveniente hacer la hipótesis de que todos los conjuntos de caracteres tienen ciertas propiedades obligatorias:

1. El tipo *char* contiene las 26 *letras* del alfabeto latino, los diez *dígitos* de la numeración árabe y cierto número de otros caracteres gráficos, como, por ejemplo, los signos de puntuación.
2. Los subconjuntos de letras y números están *ordenados* y son *coherentes*, es decir:

$$\begin{aligned} ('A' \leq x) \wedge (x \leq 'Z') &\equiv x \text{ es una letra} \\ ('0' \leq x) \wedge (x \leq '9') &\equiv x \text{ es un dígito} \end{aligned} \quad (1.5)$$

3. El tipo *char* contiene un carácter no imprimible (carácter blanco o espacio), que puede utilizarse como separador (los blancos se simbolizan por \square en la Fig. 1.1).

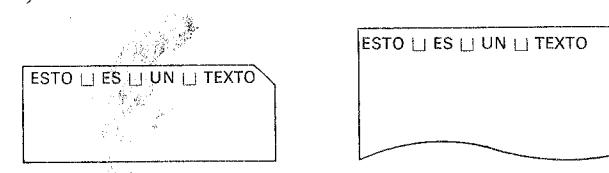


Fig. 1.1 Representación de un texto.

Para escribir programas de forma independiente del equipo de proceso de datos es particularmente importante disponer de dos funciones normalizadas de transferencia de datos entre los tipos *char* e *integer*. Se denominarán *ord(c)*, para designar el número de orden del carácter c en el conjunto que define el tipo *char* y *chr(i)* para designar el i -ésimo carácter del conjunto que define a *char*. Así pues, *chr* es la función inversa de *ord* y recíprocamente, es decir:

$$\begin{aligned} \text{ord}(\text{chr}(i)) &= i \quad (\text{si está definida } \text{chr}(i)) \\ \text{chr}(\text{ord}(c)) &= c \end{aligned} \quad (1.6)$$

Especialmente importantes son las funciones:

$$\begin{aligned} f(c) &= \text{ord}(c) - \text{ord}('0') = \text{posición de } c \text{ entre los dígitos} \\ g(i) &= \text{chr}(i + \text{ord}('0')) = \text{dígito } i \end{aligned} \quad (1.7)$$

Por ejemplo, $f('3') = 3$, $g(5) = '5'$. f es la función inversa de g y recíprocamente, es decir,

$$\begin{aligned} f(g(i)) &= i \quad (0 \leq i \leq 9) \\ g(f(c)) &= c \quad ('0' \leq c \leq '9') \end{aligned} \quad (1.8)$$

Estas funciones de transferencia se utilizan para convertir la representación interna de los números en secuencias de dígitos y recíprocamente. De hecho, representan estas conversiones al nivel más elemental, es decir, el de un dígito individual.

1.5. TIPOS SUBCAMPO

Ocurre a menudo que una variable toma valores de cierto tipo que están comprendidos en un intervalo específico. Esto puede expresarse definiendo la variable como perteneciente a un tipo *subcampo* de acuerdo con el formato:

type $T = \text{min} \dots \text{max}$	(1.9)
--	-------

siendo min y max los límites del intervalo.

EJEMPLOS

```

type año = 1900 .. 1999
type letra = 'A' .. 'Z'
type dígito = '0' .. '9'
type oficial = teniente .. general

```

Dadas las variables

```

var y: año
var L: letra

```

las asignaciones $y := 1973$ y $L := 'W'$ son admisibles, pero $y := 1291$ y $L := '9'$ no lo son. La validez de tales asignaciones no puede ser verificada por un compilador a menos que el valor a ser asignado sea el de una constante o una variable del mismo tipo. Sin embargo, la validez de asignaciones del tipo de

$y := i$ y $L := c$

siendo i del tipo *integer* y c del tipo *char*, sólo puede ser comprobada durante la ejecución del programa. La práctica ha demostrado que los sistemas que hacen estas comprobaciones son muy útiles para el desarrollo de programas. El uso que estos sistemas hacen de la información redundante para detectar posibles errores es, se insiste, la razón principal para utilizar lenguajes de alto nivel.

1.6. LA ESTRUCTURA ARRAY

El array es, probablemente, la estructura de datos más conocida, debido a que en muchos lenguajes, incluyendo FORTRAN y ALGOL 60, es la única estructura explícitamente disponible. Un array es una estructura *homogénea*; está constituida por componentes, todos ellos del mismo tipo, llamado *tipo base*. El array se denomina también estructura de *acceso aleatorio*; todos sus componentes pueden seleccionarse arbitrariamente y son igualmente accesibles. Para designar un componente aislado, el nombre de la estructura total se amplía con el denominado *índice* de selección del componente. El índice debe ser un valor del tipo definido como el *tipo índice* del array. Por tanto, la definición de un tipo $\text{array } T$ especifica tanto un tipo base T_0 como un tipo índice I .

(1.9)

type $T = \text{array}[I] \text{ of } T_0$	(1.10)
--	--------

EJEMPLOS

```

type Fila = array [1 .. 5] of real
type Tarjeta = array [1 .. 80] of char
type alfa = array [1 .. 10] of char

```

Un valor particular de una variable definida como

var $x: Fila$

en el que cada valor componente satisface la ecuación $x_i = 2^{-i}$ puede visualizarse en la forma mostrada por la Fig. 1.2.

x_1	0.5
x_2	0.25
x_3	0.125
x_4	0.0625
x_5	0.03125

Fig. 1.2 Array de tipo *Fila*.

Un valor estructurado x de tipo T con valores componentes c_1, \dots, c_n se puede designar por un *constructor* de array y una instrucción de asignación:

$x := T(c_1, \dots, c_n)$ (1.11)

El operador inverso del constructor es el *selector*. Este selecciona un componente individual del array. Dada una variable array x la notación de selector se construye con el nombre del array ampliado con el índice i del componente a seleccionar:

$x[i]$

(1.12)

La forma habitual de operar con arrays, en especial con los de dimensiones importantes, es actualizar componentes aislados en forma selectiva en lugar de construir de nuevo el conjunto de valores estructurados. Esto se expresa considerando una variable de tipo array como la representación de un array de variables componentes y permitiendo asignaciones a componentes individuales.

EJEMPLO

$$x[i] := 0.125$$

Aunque la actualización selectiva produce el cambio de un único valor componente, desde un punto de vista conceptual esto debe entenderse también como un cambio del valor de la estructura compuesta total.

El hecho de que los índices del array, es decir, los «nombres» de los componentes del array tengan que ser de un tipo (escalar) de datos definido, tiene una consecuencia muy importante: los índices pueden calcularse, es decir, puede ponerse, en lugar de un índice constante, un índice definido por una expresión. Esta expresión debe evaluarse, y su resultado determina el componente seleccionado. Esta generalización no sólo proporciona una herramienta de programación muy significativa y potente, sino que, al mismo tiempo, da ocasión a uno de los errores de programación más frecuentes: el valor resultante puede estar fuera del intervalo especificado como campo de variación de los índices del array. Se supondrá que los equipos de proceso proporcionarán mensajes adecuados en el caso de tales accesos erróneos a componentes inexistentes del array.

Normalmente, un tipo índice será escalar, es decir, un tipo no estructurado en el que estará definida una relación de orden. Si el tipo base de un array está también ordenado, se da de forma natural, sobre este tipo array, una relación de orden. Esta relación de orden entre dos arrays estará determinada por los dos componentes desiguales, correspondientes, con menor índice. Esto se expresa formalmente de la manera siguiente:

Dados dos arrays x e y , la relación $x < y$ se verifica si, y solo si, existe un índice k tal que $x[k] < y[k]$ y $x[i] = y[i]$ para todo $i < k$. (1.13)

Así, por ejemplo:

$$(2, 3, 5, 7, 9) < (2, 3, 5, 7, 11) \\ \text{'LABEL'} < \text{'LIBEL'}$$

En la mayoría de las aplicaciones, sin embargo, no se presupone que exista una relación de orden en los tipos array.

El cardinal de un tipo estructurado es el producto de los cardinales de sus componentes. Como todos los componentes de un tipo array A son del mismo tipo base B , se tiene:

$$\text{card}(A) = (\text{card}(B))^n \quad (1.14)$$

siendo $n = \text{card}(I)$, e I el tipo índice del array.

La siguiente parte de programa muestra el uso de un selector de array. El objetivo de este programa es encontrar el mínimo índice i de un componente con

valor x . La búsqueda se realiza mediante una inspección secuencial (*scan*) del array a , declarado en la forma:

```
var a: array [1 .. N] of T; {N > 0}
i := 0;
repeat i := i + 1 until (a[i] = x) ∨ (i = N);
if a[i] ≠ x then «no está este elemento en a»
```

(1.15)

Puede realizarse una variante de este programa utilizando la técnica del *centinela* apostado al final del array. El objetivo de este centinela es permitir una simplificación de la condición que señala el final de la iteración.

```
var a: array [1 .. N + 1] of T;
i := 0; a[N + 1] := x;
repeat i := i + 1 until a[i] = x;
if i > N then «no está este elemento en a»
```

(1.16)

La asignación $a[N + 1] := x$ es un ejemplo de *actualización selectiva*, es decir, de alteración de un componente particular de una variable estructurada. La condición esencial que se verifica, cualquiera que sea el número de veces que se repita la instrucción $i := i + 1$, es

$$a[j] \neq x, \quad \text{para } j = 1, \dots, i - 1$$

y esto en ambas versiones (1.15) y (1.16). Esta condición se denomina por ello un *invariante del ciclo*.

La búsqueda puede, desde luego, acelerarse considerablemente si los elementos están ya ordenados. En este caso el método de trabajo más usual es el de división repetida en partes iguales del intervalo en que debe buscarse el elemento. Este método se conoce con el nombre de *búsqueda binaria* o de *bisección* y se muestra en (1.17). En cada iteración se divide en partes iguales el intervalo entre los índices i y j . El número necesario de comparaciones es, por ello, como mucho $\lceil \log_2(N) \rceil$.

```
i := 1; j := N;
repeat k := (i + j) div 2;
if x > a[k] then i := k + 1 else j := k - 1
until (a[k] = x) ∨ (i > j)
```

(1.17)

En este caso la condición invariante a la entrada de la instrucción repetitiva es

$$a[h] < x \quad \text{para } h = 1 \dots i - 1
a[h] \geq x \quad \text{para } h = j + 1 \dots N$$

Por lo tanto, si el programa termina con $a[k] \neq x$, ello implica que no existe ningún $a[h] = x$ con $1 \leq h \leq N$.

Los elementos constituyentes de un array puede estar a su vez estructurados. Una variable de tipo array cuyos componentes son también arrays se denomina *matriz*. Por ejemplo,

M: array[1 .. 10] of Fila

es un array formado por diez componentes (filas), cada una constituida por cinco componentes de tipo real y se denomina matriz de 10×5 con componentes reales. Los selectores pueden encadenarse de manera acorde, de tal forma que

$M[i][j]$

designa el componente j de la fila $M[i]$, que es el componente i de M . Esto se abrevia usualmente en la forma

$M[i, j]$

y, en la misma idea, la declaración

M: array[1 .. 10] of array[1 .. 5] of real

puede escribirse de manera sucinta como

M: array[1 .. 10, 1 .. 5] of real

Si una determinada operación debe realizarse con *todos* los componentes de un array o con elementos consecutivos de una parte del array, puede emplearse la instrucción **for** de la forma mostrada en el ejemplo siguiente.

Supóngase que una fracción f se representa por el array d , de la forma:

$$f = \sum_{i=1}^{k-1} d_i * 10^{-i}$$

es decir, por su forma decimal con $k - 1$ dígitos. Se pide dividir f por 2. Esto se hace por repetición de la conocida operación de división de *todos* los $k - 1$ dígitos d_i , a partir de $i = 1$. El método consiste en dividir cada dígito por 2, teniendo en cuenta el posible arrastre de la posición previa, y reteniendo un posible resto para el paso siguiente [ver (1.18)].

$$\begin{aligned} r &:= 10 * r + d[i]; \\ d[i] &:= r \text{ div } 2; \\ r &:= r - 2 * d[i] \end{aligned} \quad (1.18)$$

Este proceso se aplica en el programa 1.1 para calcular una tabla de potencias negativas de 2. La repetición del proceso de división por dos para calcular $2^{-1}, 2^{-2}, \dots, 2^{-n}$ se expresa nuevamente, en forma apropiada, mediante la instrucción **for**, conduciendo así a un doble ciclo **for**, constituido por un ciclo comprendido dentro de otro.

Programa 1.1 Cálculo de las potencias de 2.

```
program potencia (output);
{representación decimal de las potencias negativas de 2}
const n = 10;
type digito = 0 .. 9;
var i,k,r: integer;
d: array [1 .. n] of digito;
begin for k := 1 to n do
begin write ('.'); r := 0;
  for i := 1 to k - 1 do
    begin r := 10 * r + d[i]; d[i] := r div 2;
      r := r - 2 * d[i]; write(chr(d[i] + ord('0')));
    end;
    d[k] := 5; writeln('5')
  end
end.
```

La salida resultante para $n = 10$ es

```
.5
.25
.125
.0625
.03125
.015625
.0078125
.00390625
.001953125
.0009765625
```

1.7. LA ESTRUCTURA REGISTRO

El método más general de obtención de tipos estructurados es yuxtaponer elementos de cualquier tipo, incluso estructurados, para obtener un tipo compuesto. Ejemplos de esta forma de proceder, en matemáticas, son los números

complejos, compuestos de dos números reales y las coordenadas de puntos compuestas de dos o más números reales, según las dimensiones del espacio definido por el sistema de coordenadas. Un ejemplo similar, en proceso de datos, es la forma de describir a las personas mediante unas pocas características relevantes, como su nombre y apellidos, así como su fecha de nacimiento, sexo y estado civil.

En matemáticas un tipo compuesto de esta forma se denomina *producto cartesiano* de los tipos que lo constituyen. Esto se debe a que el conjunto de valores definido por este tipo compuesto, está formado por todas las combinaciones posibles de valores, tomando uno de cada conjunto de valores asignado a cada tipo constituyente. Así, el número de tales combinaciones, también llamadas *n-uplos*, es el producto del número de elementos de cada conjunto constituyente, es decir, el cardinal del tipo compuesto es el producto de los cardinales de los tipos simples que lo forman.

En proceso de datos, los tipos compuestos, tales como las descripciones de personas u objetos, habitualmente aparecen en ficheros o «bancos de datos», y registran las características relevantes de una persona u objeto. Por esto la palabra *registro* se acepta de forma general para describir datos compuestos de esta forma, y se adoptará en lo que sigue esta denominación con preferencia sobre producto cartesiano.

En general, el tipo registro *T* se define de la forma siguiente:

```
type T = record s1: T1;
               s2: T2;
               ...
               sn: Tn
           end
```

(1.19)

$$\text{Card}(T) = \text{card}(T_1) * \dots * \text{card}(T_n)$$

EJEMPLOS

```
type Complejo = record re: real;
               im: real
           end

type Fecha = record dia: 1 .. 31;
               mes: 1 .. 12;
               año: 1 .. 2000
           end

type Persona = record apellido: alfa;
```

```
nombre: alfa;
nacimiento: Fecha;
sexo: (varon, hembra);
ecivil: (soltero, casado,
        viudo, divorciado)
```

end

Mediante un *constructor de registro* pueden formarse valores de tipo *T* y, subsiguientemente, ser asignados a una variable de este tipo:

$$x := T(x_1, x_2, \dots, x_n) \quad (1.20)$$

en donde los *x_i* son valores de los tipos constituyentes *T_i*.

Dadas las variables registro

```
z: Complejo
f: Fecha
p: Persona
```

puede asignarse valores, por ejemplo, de la forma siguiente (ver Fig. 1.3):

```
z := Complejo (1.0, -1.0)
f := Fecha (1,4,1973)
p := Persona ('WIRTH', 'CHRIS', Fecha (18,1,1966), varon, soltero)
```

Complejo z	Fecha f	Persona p
1.0 -1.0	1 4 1973	WIRTH CHRIS 18 1 1966 varón soltero

Fig. 1.3 Registros de los tipos *complejo*, *fecha* y *persona*.

Los identificadores *s₁*, ..., *s_n*, introducidos mediante una definición de tipo registro, son los nombres dados a las componentes individuales del tipo, y se utilizan como *selectores de registro* aplicados a variables con esta estructura. Dada una variable *x*: *T* su componente *i* se designa por

x.s_i

(1.21)

La actualización selectiva de x se lleva a cabo utilizando la misma notación a la izquierda de una instrucción de asignación:

$$x \cdot s_i := x_i$$

en donde x_i es un valor (o expresión) de tipo T_i .

Dadas las variables registro

z : Complejo
 f : Fecha
 p : Persona

los siguientes son selectores de componentes de z , f y p :

$z \cdot im$	(de tipo real)
$f \cdot mes$	(de tipo 1 .. 12)
$p \cdot apellido$	(de tipo alfa)
$p \cdot nacimiento$	(de tipo Fecha)
$p \cdot nacimiento \cdot dia$	(de tipo 1 .. 31)

El ejemplo del tipo *Persona* muestra cómo un componente de un tipo registro puede a su vez estar estructurado. Así, pues, los selectores pueden encadenarse. Naturalmente los diferentes tipos estructurantes pueden utilizarse de manera que unos formen parte de otros. Por ejemplo, el componente i de un array a que a su vez forma parte de una variable registro r se denomina:

$$r \cdot a[i],$$

y el componente con nombre de selector s del elemento i de un array estructurado en forma de registro, se designa por:

$$a[i] \cdot s$$

El producto cartesiano se caracteriza por contener *todas* las combinaciones de elementos de los tipos que lo constituyen; pero hay que hacer notar, a efectos de aplicación práctica, que no todas son «legales», es decir, significativas. Por ejemplo, el tipo *Fecha* definido anteriormente, contiene los valores

$$(31,4,1973) \quad y \quad (29,2,1815)$$

que son, ambos, fechas de días que nunca tuvieron lugar. Por tanto, la definición de este tipo no refleja la situación real, pero es suficientemente aproximada a efectos prácticos, y corresponde al programador asegurarse de que los valores

carentes de significado no se produzcan durante la ejecución del programa.

A continuación se presenta una parte de un programa que muestra cómo utilizar las variables registro. Tiene por objeto contar el número de «Personas» representadas por la variable array a , que son simultáneamente femeninas y solteras.

```
var a: array[1 .. N] of Persona;
    contador: integer;
    contador := 0;
for i := 1 to N do
    if (a[i] · sexo = hembra) ∧ (a[i] · ecivil = soltero) then
        contador := contador + 1
```

(1.22)

En este caso la condición invariante del ciclo es:

$$\text{contador} = C(i)$$

en donde $C(i)$ es el número de personas solteras, femeninas, que forman parte del subconjunto a_1, \dots, a_i .

Otra forma de escribir la última instrucción utilizando una construcción denominada *instrucción with* es:

```
for i := 1 to N do
    with a[i] do
        if (sexo = hembra) ∧ (ecivil = soltero) then
            contador := contador + 1
```

(1.23)

El significado de **with r do s** es que los nombres de los selectores de tipo de la variable r pueden usarse sin prefijo, dentro de la instrucción s , referidos a la variable r . La instrucción *with*, por ello, sirve tanto para abreviar el texto del programa, como para evitar la doble evaluación de la dirección de memoria de la parte indexada $a[i]$.

En un ejemplo, más adelante, se supone que (posiblemente para agilizar su búsqueda) ciertos grupos de personas en el array a están conectados entre ellos. La información de conexión se representa por un nuevo componente en la estructura del registro *Persona* denominado *enlace*. Los enlaces conectan registros en cadena de forma que el siguiente y el precedente de cada persona pueden encontrarse fácilmente. Esta técnica de enlace tiene la interesante propiedad de que permite recorrer la cadena en ambas direcciones almacenando un sólo número en cada registro. Esta técnica funciona de la forma que se indica a continuación.

Se supone que los índices de tres elementos consecutivos de la cadena son i_{k-1} , i_k , i_{k+1} . Como valor para enlace del elemento k se elige $i_{k+1} - i_{k-1}$. La forma de recorrer la cadena hacia adelante se determina calculando i_{k+1} a partir de las variables índice $x = i_{k-1}$ e $y = i_k$ en la forma:

$$i_{k+1} = x + a[y] \cdot \text{enlace}$$

En cambio, la forma de recorrer la cadena hacia atrás, se determina calculando i_{k-1} a partir de $x = i_{k+1}$ e $y = i_k$ en la forma:

$$i_{k-1} = x - a[y] \cdot \text{enlace}$$

Como ejemplo se incluye la forma de enlazar las personas del mismo sexo, en la Tabla 1.2.

	Nombre	Sexo	Enlace
1	Carolina	H	2
2	Antonio	V	2
3	Tina	H	5
4	Roberto	V	3
5	Julián	V	3
6	Genoveva	H	5
7	Raimundo	V	5
8	Maria	H	3
9	Ana	H	1
10	Matias	V	3

Tabla 1.2. Array con elementos del tipo *Persona*.

Las estructuras registro y array tienen en común la propiedad de que ambas son estructuras de acceso directo. El registro es más general en el sentido de que no requiere que los tipos que lo forman sean idénticos. En cambio, el array presenta mayor flexibilidad al permitir que sus selectores de componentes sean valores computables (representados por expresiones), mientras que los selectores de componentes de un registro deben ser identificadores fijos, declarados en la definición de tipo de registro.

1.8. VARIANTES DE LAS ESTRUCTURAS REGISTRO

En la práctica es a menudo conveniente y natural considerar dos tipos de datos como *variantes* del mismo tipo. Por ejemplo, el tipo *coordenada* del párrafo anterior puede considerarse como la unión de sus dos variantes de coordenadas, cartesianas y polares, que están constituidas, respectivamente, por *a) dos longitudes y b) una longitud y un ángulo*. Para identificar la variante adoptada por una variable, se introduce un tercer componente denominado *discriminante de tipo* o *campo indicador*.

```
type Coordenada =
  record case clase: (Cartesiana, polar) of
    Cartesiana: (x, y: real);
```

```
polar: (r: real; φ: real)
```

```
end
```

Aquí el nombre del discriminante es *clase*, y los nombres de las coordenadas son *x, y* en el caso de valores cartesianos o *r, φ* en el caso de polares.

El conjunto de valores designados por el tipo *Coordenada* es la *unión* de los correspondientes a los dos tipos

```
T1 = (x, y: real)
```

```
T2 = (r: real; φ: real)
```

su número cardinal es la suma de los correspondientes a *T₁* y *T₂*.

Sin embargo, muy frecuentemente no se unen dos tipos enteramente distintos sino, más bien, tipos con algunos componentes comunes. Esta es la situación que dio origen al término estructura *registro variante*. Un ejemplo lo constituye el tipo *Persona* definido en el párrafo anterior en el que las características relevantes a registrar en un fichero dependen del sexo de la persona. Por ejemplo, para un hombre, su peso y la tenencia o no de barba pueden ser de interés en determinadas situaciones, en cambio, para una mujer pueden considerarse significativas tres medidas características (sin embargo, su peso puede ser confidencial). De las consideraciones anteriores resulta la definición de tipo siguiente:

```
type Persona =
  record apellido, nombre: alfa;
    nacimiento: Fecha;
    ecivil: (soltero, casado, viudo, divorciado);
    case sexo: (varon, hembra) of
      varon: (peso: real;
               barbudo: boolean);
      hembra: (tamaño: array[1..3] of integer)
  end
```

La forma general de la definición de un tipo registro variante es

```
type T =
  record s1: T1; ...; sn-1: Tn-1;
    case sn: Tn of
      c1: (s1,1: T1,1; ...; s1,n1: T1,n1);
      ...
      cm: (sm,1: Tm,1; ...; sm,nm: Tm,nm)
  end
```

(1.24)

Los s_i y s_{ij} son los nombres de selectores de componentes cuyos tipos son T_i , T_{ij} y s_n es el nombre del campo discriminante cuyo tipo es T_n . Las constantes c_1, \dots, c_m designan valores del tipo (escalar) T_n . Una variable x de tipo T consta de los componentes:

$$x.S_1, x.S_2, \dots, x.S_n, x.S_{k,1}, \dots, x.S_{k,n_k}$$

si y solo si el valor de $x \cdot s_n = c_k$. Los componentes $x \cdot s_1, \dots, x \cdot s_n$ constituyen la parte común de todas las m variantes.

Por consiguiente, debe considerarse un error importante de programación el uso de un selector de componente $x \cdot s_{k,h}$ ($1 \leq h \leq n_k$) cuando $x \cdot s_k \neq c_k$ y (con referencia al tipo *Persona* definido anteriormente) conduciría a consultar si una señora tiene barba o (en el caso de actualización selectiva) a obligarla a tenerla.

La utilización de registros variantes debe hacerse, por tanto, con sumo cuidado y la mejor forma de hacerlo es agrupar las operaciones correspondientes a cada variante en una instrucción selectiva, llamada instrucción *case*, cuya estructura es una imagen de la de definición del tipo registro variante.

```
case x.sn of
  c1: S1;
  c2: S2;
  ...
  cm: Sm
end
```

(1.25)

S_k representa las instrucciones correspondientes al caso de que x adopte la forma de la variante k , es decir, se selecciona su ejecución solamente cuando el campo discriminante $x \cdot s_n$ toma como valor c_k . Por tanto, es relativamente fácil ponerse a salvo de errores en el uso de nombres de selectores comprobando que cada S_k contiene como selectores únicamente

$x.s_1 \dots x.s_{n-1}$

y

$x.s_{k,1} \dots x.s_{k,n_k}$

La parte de programa siguiente tiene por objeto calcular la distancia entre dos puntos A y B dados por las variables a y b del tipo registro variante *Coordenada*. El proceso de cálculo difiere según las cuatro combinaciones posibles de coordenadas polares y cartesianas (ver Fig. 1.4).

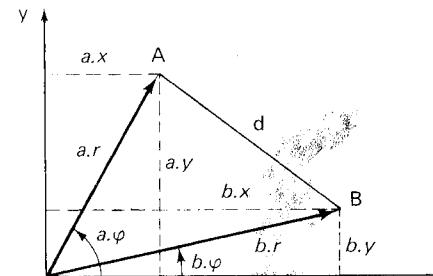


Fig. 1.4 Coordenadas polares y cartesianas.

case a · clase of

Cartesiana: **case b · clase of**

Cartesiana: $d := \sqrt{(a.x - b.x)^2 + (a.y - b.y)^2}$;

Polar: $d := \sqrt{\sqrt{(a.x - b.r * \cos(b.\phi))^2 + (a.y - b.r * \sin(b.\phi))^2}}$

end;

case b · clase of

Cartesiana: $d := \sqrt{(a.r * \sin(a.\phi) - b.x)^2 + (a.r * \cos(a.\phi) - b.y)^2}$;

Polar: $d := \sqrt{\sqrt{(a.r)^2 + (b.r)^2} - 2 * a.r * b.r * \cos(a.\phi - b.\phi)}$

end

end

1.9. LA ESTRUCTURA CONJUNTO (SET)

La tercera estructura fundamental, además del array y el registro, es la *estructura conjunto* (*set*). Se define de la siguiente forma:

type T = set of T₀

(1.26)

Los valores posibles de una variable x del tipo T son conjuntos de elementos de T_0 . El conjunto de todos los subconjuntos formados por elementos de un conjunto T_0 se denomina *conjunto potencia* o *conjunto de las partes* de T_0 . El tipo T , por tanto, es el conjunto potencia de su tipo base T_0 .

HJEMPLOS

type entset = set of 0 .. 30

type charset = set of char

type estadocinta = set of excepcion

El segundo ejemplo se basa en el conjunto normalizado de caracteres definido por el tipo *char*; el tercer ejemplo se basa en un conjunto de condiciones de excepción que pueden definirse mediante un tipo escalar

`type excepcion = (descargada, manual, paridad, desalineada)`

que describe los distintos estados excepcionales en que puede encontrarse una cinta magnética. Dadas las variables

```
es : entset
cs : charset
t : array [1 .. 6] of estadocinta
```

pueden construirse y asignarse a ellas, valores de los distintos tipos, por ejemplo, de la forma siguiente†:

```
es := [1, 4, 9, 16, 25]
cs := ['+', '-', '*', '/']
t[3] := [manual]
t[5] := []
t[6] := [descargada .. desalineada]
```

Aquí el valor asignado a $t[3]$ es el conjunto constituido por el elemento único *manual*; a $t[5]$ se asigna el conjunto vacío, significando que la quinta unidad de *cinta* vuelve al estado operacional (no-excepcional), mientras a $t[6]$ se asigna el conjunto de las cuatro excepciones.

El cardinal de un tipo conjunto T es

$$\text{card}(T) = 2^{\text{card}(T_0)} \quad (1.27)$$

Esto puede deducirse fácilmente a partir del hecho de que cada uno de los $\text{card}(T_0)$ elementos de T_0 puede representarse por uno de los dos valores «presente» o «ausente» y de que todos los elementos son independientes unos de otros. Evidentemente, es esencial, para una implantación eficiente y económica, que el tipo base sea no sólo finito sino razonablemente pequeño.

Los operadores elementales que se definen en las estructuras de tipo conjunto son los siguientes:

- * intersección de conjuntos
- + unión de conjuntos
- diferencia de conjuntos
- in pertenencia a un conjunto

Las operaciones de intersección y unión de conjuntos se llaman frecuentemente *producto* y *suma de conjuntos*, respectivamente. Las prioridades de los ope-

† Contrariamente a la notación convencional se utilizan corchetes en lugar de llaves para los conjuntos. Se reservan las llaves para delimitar comentarios en los programas.

radores de conjuntos se definen de acuerdo con el siguiente criterio: el operador de intersección tiene prioridad sobre los de unión y diferencia que, a su vez, tienen prioridad sobre el operador de pertenencia; este último se considera un operador de relación. A continuación se presentan ejemplos de expresiones con conjuntos y sus equivalentes con paréntesis explícitos:

$$\begin{aligned} r * s + t &= (r * s) + t \\ r - s * t &= r - (s * t) \\ r - s + t &= (r - s) + t \\ x \text{ in } s + t &= x \text{ in } (s + t) \end{aligned}$$

El primer ejemplo que se presenta de aplicación de la estructura conjunto es el programa de un analizador léxicográfico («scanner») sencillo de un compilador. Se supone que el objetivo de este analizador es traducir una secuencia de caracteres a una secuencia de unidades de texto del lenguaje a compilar, llamadas «piezas» o *símbolos*. El analizador se representará por un procedimiento que cada vez que es llamado lee un número suficiente de caracteres de entrada para generar un nuevo símbolo de salida. Las reglas particulares de traducción son:

1. El conjunto de símbolos de salida está formado por los elementos *identificador*, *numero*, *menoroigual*, *mayoroigual*, *tomaelvalorde*, y otros que corresponden a varios caracteres aislados tales como +, -, *, etc.
2. El símbolo *identificador* se genera al leer una secuencia de letras y dígitos que comienza con una letra.
3. El símbolo *numero* se genera al leer una secuencia de dígitos.
4. Los símbolos *menoroigual*, *mayoroigual* y *tomaelvalorde* se generan al leer los segmentos, parejas de caracteres, < =, > =, : =.
5. Los blancos y finales de línea se saltan.

Se cuenta con el procedimiento básico *read(x)* que extrae el carácter siguiente de la secuencia de entrada y lo asigna a la variable *x*. El símbolo de salida resultante se asigna a una variable global llamada *sym*. Además, están las variables globales *id* y *num* cuyo objetivo resulta evidente en el programa 1.2 y *ch* que representa el carácter de la secuencia de entrada en fase de análisis. *S* designa una aplicación del conjunto de caracteres sobre el de símbolos, es decir, un array de símbolos con un dominio de índices formado por caracteres que no son ni dígitos ni letras. El uso de conjuntos de caracteres demuestra cómo un analizador léxicográfico puede programarse independientemente del orden que exista entre los caracteres que forman los textos.

El segundo ejemplo consiste en la construcción de un horario escolar. Se supone que *M* estudiantes han elegido entre *N* asignaturas. Debe construirse

un horario de forma que se programen determinadas asignaturas para ser explícadas a la misma hora sin que haya conflictos [1.1].

Programa 1.2. Analizador lexicográfico («Scanner»).

```

var ch: char;
sym: simbolo;
num: integer;
id: record
  k: 0 .. maxk;
  a: array [1 .. maxk] of char
end;
procedure analizador;
  var ch1: char;
begin {saltar blancos}
  while ch = ' ' do read(ch);
  if ch in ['A' .. 'Z'] then
    with id do
      begin sym := identificador; k := 0;
      repeat if k < maxk then
        begin k := k + 1; a[k] := ch
        end;
        read(ch)
      until not(ch in ['A' .. 'Z', '0' .. '9'])
      end else
    if ch in ['0' .. '9'] then
      begin sym := numero; num := 0;
      repeat num := 10 * num + ord(ch) - ord('0');
      read(ch)
      until not(ch in ['0' .. '9'])
      end else
    if ch in ['<', ':', '>'] then
      begin ch1 := ch; read(ch);
      if ch = '=' then
        begin
          if ch1 = '<' then sym := menorigual else
          if ch1 = '>' then sym := mayorigual else sym := tomaelvalorde;
          read(ch)
        end
        else sym := S[ch1]
      end else
    begin {otros símbolos}
      sym := S[ch]; read(ch)
    end
  end {analizador}

```

(1.28)

```

type asignatura = 1 .. N;
estudiante = 1 .. M;
seleccion = set of asignatura;
var s: asignatura;
i: estudiante;
matricula: array [estudiante] of seleccion;
conflicto: array [asignatura] of seleccion;
{Determinación de los conjuntos de asignaturas conflictivas
a partir de las asignaturas en que se matricula cada estudiante}
for s := 1 to N do conflicto[s] := [];
for i := 1 to M do
  for s := 1 to N do
    if s in matricula[i] then
      conflicto[s] := conflicto[s] + matricula[i]

```

(Obsérvese que *s* in *conflicto*[*s*] es una consecuencia de este algoritmo.)

En general, la construcción de un horario es un problema combinatorio muy difícil, donde debe tomarse una decisión teniendo en cuenta muchos factores y con abundantes restricciones. En este ejemplo se simplificará el problema de forma drástica sin plantearse la búsqueda de una solución de horario realista.

En primer lugar se constata que para obtener clases simultáneas aceptables es preciso basar las decisiones en el conjunto de datos obtenidos de las matrículas individuales de los alumnos, es decir, en la enumeración de asignaturas que no puedan explicarse simultáneamente. Por ello, se programa en primer lugar un proceso de reducción de datos basado en las siguientes declaraciones y en el convenio de que los estudiantes se numeran de 1 a *M* y las asignaturas de 1 a *N*.

La tarea principal ahora consiste en construir un horario, es decir, una lista de sesiones, cada sesión formada por una selección de asignaturas que no están en conflicto. Del conjunto de todas las asignaturas se eligen subconjuntos de asignaturas aceptables, no conflictivas, suprimiéndolas de una variable conjunto denominada *remanente*, hasta que este conjunto de asignaturas remanentes quede vacío.

(1.29)

```

var k: integer;
remanente, sesion: seleccion;
horario: array [1 .. N] of seleccion;
k := 0; remanente := [1 .. N];
while remanente ≠ [] do
  begin sesion := siguiente seleccion aceptable;
  remanente := remanente - sesion;
  k := k + 1; horario[k] := sesion
  end

```

¿Cómo se realiza una «siguiente selección aceptable»? Al iniciar el proceso puede elegirse cualquier asignatura del conjunto de las remanentes. Subsiguientemente, la elección de nuevos candidatos puede estar restringida al conjunto de asignaturas remanentes que no entran en conflicto con las inicialmente seleccionadas. Este conjunto se llamará de *prueba*. Cuando se investiga un candidato del conjunto de prueba, se constata que su elección depende de que la intersección del conjunto de asignaturas seleccionado y el conjunto de las que entran en conflicto con el candidato esté o no vacía. Esto conduce a la siguiente elaboración de la instrucción «*sesion* := siguiente selección aceptable»:

```

var s, t: asignatura;
      prueba: selección;
begin s := 1;
      while  $\neg(s \text{ in } \text{remanente})$  do s := s + 1;
      sesion := [s]; prueba := remanente - conflicto[s];
      for t := 1 to N do
          if t in prueba then
              begin if conflicto[t] * sesion = [] then
                  sesion := sesion + [t]
              end
      end

```

(1.30)

Evidentemente esta solución para seleccionar sesiones «aceptables» generará horarios no necesariamente óptimos en algún aspecto específico. Puede darse en algún caso no afortunado la circunstancia de que el número de sesiones resultantes sea tan amplio como el de asignaturas, incluso aunque fuera posible programar ciertas clases simultáneamente.

1.10. REPRESENTACION DE LAS ESTRUCTURAS ARRAY, REGISTRO Y CONJUNTO

La esencia del uso de abstracciones en programación es que un programa puede concebirse, comprenderse y verificarse en base a las leyes que rigen aquéllas y que no es necesario preocuparse de la forma en que se implantan y representan en un determinado computador. De todas formas es una ayuda para el éxito del programador la comprensión de las técnicas de uso más generalizado para representar los conceptos básicos de las abstracciones de programación, tales como las estructuras fundamentales de datos. Es una ayuda en el sentido de que capacita al programador para tomar decisiones sensatas sobre el diseño del programa y los datos teniendo en cuenta no sólo las propiedades abstractas de las estructuras sino también las de su realización concreta en los computado-

Estructura	Declaración	Selector	Acceso a los componentes por	Tipos componentes	Cardinalidad
Array	a: array[I] of T ₀	a[i] (i ∈ I)	Selector con índice calculable i	Todos idénticos (T ₀)	card(T ₀) ^{card(I)}
Registro	r: record s ₁ : T ₁ ; s ₂ : T ₂ ; ... s _n : T _n	r.s (s ∈ {s ₁ , ..., s _n })	Selector con nombre declarado de componente s	Pueden diferir individualmente	$\prod_{i=1}^n \text{card}(T_i)$
Conjunto	s: set of T ₀	Ninguno	Test de pertenencia con el operador de relación	Todos idénticos (y de tipo escalar T ₀)	2 ^{card(T₀)}

Tabla 1.3 Estructuras de datos fundamentales.

res, teniendo en cuenta las capacidades y limitaciones de un determinado computador.

El problema de la representación de datos es el de definir una aplicación de la estructura abstracta sobre la memoria del computador. Las memorias de los computadores son, en primera aproximación, arrays de celdas de almacenamiento individual denominadas *palabras*. Los índices de las palabras se llaman direcciones.

var memoria: array [direccion] of palabra (1.31)

Los cardinales de los tipos *direccion* y *palabra* varían de unos computadores a otros. La gran variabilidad del cardinal de la palabra es un problema. Su logaritmo se llama *tamaño de la palabra*, porque es el número de bits que contiene una celda de memoria.

1.10.1. Representación de los arrays

Una representación de una estructura array es una aplicación del array (abstracto), con componentes de tipo *T*, sobre la memoria, que a su vez es un array con componentes de tipo *palabra*.

El array debe aplicarse de tal forma que el cálculo de direcciones de sus componentes sea lo más simple (y, por tanto, eficiente) posible. La dirección o índice de memoria *i* del componente *j* del array se calcula por la función de aplicación lineal

$$i = i_0 + j * s \quad (1.32)$$

en donde *i*₀ es la dirección del primer componente y *s* es el número de palabras que «ocupa» cada componente. Dado que la palabra es por definición la unidad de memoria más pequeña a la que puede accederse individualmente, es muy deseable, evidentemente, que *s* sea un número entero, en el caso más sencillo *s* = 1. Si *s* no es entero (que es el caso normal) se redondea, habitualmente, hasta el entero mayor más próximo $\lceil s \rceil$. En este caso, cada componente del array ocupa $\lceil s \rceil$ palabras y, por tanto, $\lceil s \rceil - s$ palabras no se usan (ver Figs. 1.5 y 1.6). Este redondeo del número de palabras necesarias hasta el entero más próximo, se llama *ajuste*. El factor de «utilización de memoria», *u*, es el cociente de la cantidad mínima de memoria necesaria para representar una estructura y la cantidad realmente utilizada:

$$u = \frac{s}{s'} = \frac{s}{\lceil s \rceil} \quad (1.33)$$

Como a la hora de implantar se tiende a que el factor de utilización de memoria sea lo más próximo posible a 1, y dado que el acceso a zonas parciales de las pa-

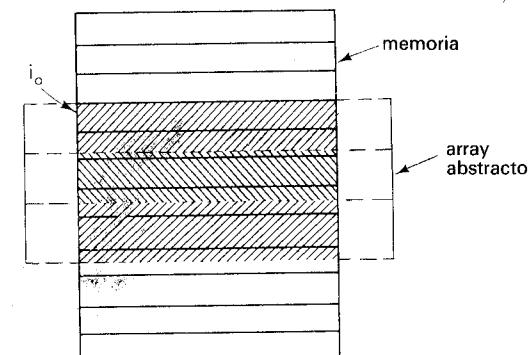


Fig. 1.5 Aplicación de un array sobre la memoria.

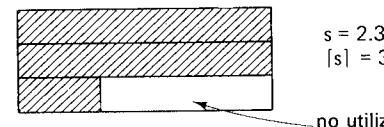


Fig. 1.6 Representación del ajuste de un registro.

bras es un proceso farragoso y relativamente ineficiente, hay que llegar a un compromiso. Para ello, deben hacerse las siguientes consideraciones:

1. El ajuste disminuye la utilización de memoria.
2. La omisión del ajuste puede necesitar el acceso ineficiente a zonas parciales de la palabra.
3. El acceso a las zonas parciales de la palabra, puede dar lugar a que aumente el programa compilado y, por ello, contrarrestar el beneficio obtenido por la omisión del ajuste.

De hecho, las consideraciones 2 y 3 son frecuentemente tan dominantes que los compiladores ajustarán siempre automáticamente. Se observa que el factor de utilización será siempre $u > 0,5$, si $s > 0,5$. Sin embargo, si $s \leq 0,5$, el factor de utilización puede incrementarse de manera significativa almacenando más de un componente del array en cada palabra. Esta técnica se denomina *empaquecido*. Si se empaquetan *n* componentes en una palabra, el factor de utilización es (ver Fig. 1.7).

$$u = \frac{n \cdot s}{\lceil n \cdot s \rceil} \quad (1.34)$$



Fig. 1.7 Empaquetado de seis componentes en una palabra.

Para acceder al componente i de un array empaquetado es preciso calcular la dirección j de la palabra en que está localizado y su posición k dentro de la palabra.

$$\begin{aligned} j &= i \text{ div } n \\ k &= i \text{ mod } n = i - j * n \end{aligned} \quad (1.35)$$

En la mayoría de los lenguajes de programación el programador no tiene control sobre la representación de las estructuras abstractas de datos. Sin embargo, debería ser posible indicar el deseo de empaquetar, al menos en los casos en que quepa más de un componente por palabra, es decir, cuando pueda conseguirse una ganancia en economía de memoria de factor 2 o superior. Se adopta el convenio de indicar este deseo de empaquetar, precediendo el símbolo **array** (o **record**), en la declaración, por el símbolo **packed**.

EJEMPLO

```
type alfa = packed array [1 .. n] of char
```

Esta posibilidad es particularmente valiosa en computadores con palabras grandes y con accesibilidad a campos parciales de la palabra relativamente adecuada. La propiedad esencial de este prefijo es que de ninguna forma cambia el significado (o validez) de un programa. Esto significa que la elección de una representación alternativa puede indicarse fácilmente con la garantía implícita de que el significado del programa permanece invariable.

El coste de acceso a los componentes de un array empaquetado puede reducirse drásticamente si todo el array se empaqueta (o desempaqueta) al mismo tiempo. La razón es que puede hacerse un barrido secuencial eficiente sobre todo el array, sin que sea necesario evaluar una complicada función para cada componente individual. Por ello se postula la existencia de dos procedimientos normalizados **pack** y **unpack**, definidos de la forma que se indica a continuación. Se suponen las variables

```
u: array [a .. d] of T
p: packed array [b .. c] of T
```

donde $a \leq b \leq c \leq d$ son todos del mismo tipo escalar. De acuerdo con esto

$$\text{pack}(u, i, p), \quad (a \leq i \leq b - c + d) \quad (1.36)$$

es equivalente a

$$p[j] := u[j + i - b], \quad j = b \dots c$$

y

$$\text{unpack}(p, u, i), \quad (a \leq i \leq b - c + d) \quad (1.37)$$

es equivalente a

$$u[j + i - b] := p[j], \quad j = b \dots c$$

1.10.2. Representación de las estructuras registro

Los registros se aplican sobre la memoria (se les asigna memoria) del computador por simple yuxtaposición de sus componentes. La dirección de un componente (campo) r_i respecto de la dirección origen del registro r se llama desplazamiento del componente, k_i . Se calcula en la forma

$$k_i = s_1 + s_2 + \dots + s_{i-1} \quad (1.38)$$

siendo s_j el tamaño (en palabras) del componente j . El hecho de que todos los componentes de un array son de igual tipo tiene como consecuencia que

$$s_1 = s_2 = \dots = s_n$$

y por ello

$$k_i = s_1 + \dots + s_{i-1} = (i-1) \cdot s$$

La generalidad de la estructura registro no permite funciones lineales tan simples, normalmente, para el cálculo de los desplazamientos, y ésta es la razón principal de que los componentes de un registro deban ser seleccionables únicamente por identificadores fijos. Esta restricción tiene la deseable consecuencia de que los respectivos desplazamientos se conocen en el momento de compilación. Es muy conocida la gran eficiencia en el acceso a los campos de un registro.

Puede plantearse el problema de empaquetado si varios componentes pueden encajarse en una única palabra de memoria (ver Fig. 1.8). También aquí, el deseo de empaquetar puede indicarse en una declaración precediendo el símbolo **record** por el símbolo **packed**. Como los desplazamientos son calculables por un compilador, el desplazamiento de un componente en una palabra puede también determinarse por un compilador. Esto significa que en muchos computadores el empaquetado de registros producirá una disminución en la eficiencia del acceso considerablemente menor que la producida por el empaquetado de arrays.

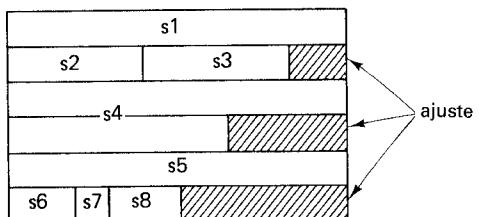


Fig. 1.8 Representación de un registro empaquetado.

1.10.3. Representación de los conjuntos

Un conjunto s se representa adecuadamente en la memoria del computador mediante su *función característica* $C(s)$. Esta es un array de valores lógicos cuyo componente i -ésimo especifica la presencia o ausencia del valor i en el conjunto. El tamaño del array está determinado por el cardinal del tipo del conjunto

$$C(s_i) \equiv (i \text{ in } s) \quad (1.39)$$

Por ejemplo, el conjunto de dígitos:

$$s = [1, 4, 8, 9]$$

se representa por la secuencia de valores lógicos F (falso) y V (verdad),

$$C(s) = (FVFFVFFFVV)$$

si el tipo base de s es el subconjunto de enteros $0 \dots 9$. En la memoria del computador, la secuencia de valores lógicos se representa por una *secuencia de bits* (ver Fig. 1.9).

S	0	1	0	0	1	0	0	0	1	1
	0	1	2	...		9				

Fig. 1.9 Representación de un conjunto mediante una secuencia de bits.

La representación de conjuntos mediante su función característica tiene la ventaja de que la obtención de la unión, intersección y diferencia de dos conjuntos puede implementarse mediante operaciones lógicas elementales. Las siguientes equivalencias, que se cumplen para todos los elementos i del tipo base de los conjuntos x e y , relacionan operaciones lógicas con operaciones entre conjuntos:

$$\begin{aligned} i \text{ in } (x \sqcup y) &\Leftrightarrow (i \text{ in } x) \vee (i \text{ in } y) \\ i \text{ in } (x * y) &\Leftrightarrow (i \text{ in } x) \wedge (i \text{ in } y) \\ i \text{ in } (x - y) &\Leftrightarrow (i \text{ in } x) \wedge \neg(i \text{ in } y) \end{aligned} \quad (1.40)$$

Estas operaciones lógicas existen en todos los computadores digitales y además operan *simultáneamente* con todos los elementos correspondientes (bits) de una palabra. De acuerdo con ello, parece que para poder implantar las operaciones básicas entre conjuntos de forma eficiente, éstos deben representarse por un número fijo, pequeño, de palabras sobre las que sea posible utilizar no sólo las operaciones lógicas básicas, sino también las de desplazamiento (*shift*). La comprobación de pertenencia a un conjunto se implanta físicamente, entonces, mediante desplazamiento único y subsiguiente operación de test del bit de signo. Por tanto, un test de la forma:

$$x \text{ in } [c_1, c_2, \dots, c_n]$$

puede implementarse de una forma mucho más eficiente que la expresión booleana convencional

$$(x = c_1) \vee (x = c_2) \vee \dots \vee (x = c_n)$$

Por tanto, la estructura conjunto debe emplearse solamente con *tipos base pequeños*. El límite de cardinalidad del tipo base, para el que puede garantizarse una implantación física razonablemente eficiente, viene determinado por la longitud de palabra del computador de que se trate y es evidente que los computadores de palabra larga son preferibles a este respecto. Si el tamaño de palabra es relativamente pequeño puede adoptarse una representación que utilice varias palabras.

1.11. LA ESTRUCTURA FICHERO SECUENCIAL

La característica común de las estructuras de datos presentadas hasta aquí, es decir, el array, el registro y el conjunto, es que su *cardinalidad es finita* (siempre que el cardinal de los tipos de sus componentes sea finito). Por ello, presentan pocas dificultades de implantación; pueden encontrarse representaciones válidas para las mismas, en cualquier computador digital.

La mayoría de las llamadas estructuras avanzadas —secuencias, árboles, grafos, etc....— se caracterizan porque su cardinalidad es infinita. Esta diferencia con las estructuras fundamentales de cardinalidad finita es de gran importancia y tiene consecuencias prácticas significativas. Como ejemplo, se define la estructura *secuencia* como sigue:

Una secuencia con tipo base T_0 es, bien la secuencia vacía, o la concatenación de una secuencia (con tipo base T_0) con un valor de tipo T_0 .

La secuencia tipo T así definida comprende infinitos valores. Cada valor en sí mismo contiene un número finito de componentes de tipo T_0 , pero este número no está acotado, es decir, para cualquier secuencia dada es posible construir otra más larga.

Consideraciones análogas pueden hacerse sobre otras estructuras «avanzadas»

de datos. La más importante es que la cantidad de memoria necesaria para representar un dato de un tipo estructural avanzado no se conoce en el momento de compilación; de hecho, puede variar durante la ejecución del programa. Esto requiere algún esquema de *asignación dinámica de memoria* en el que la memoria se vaya ocupando a medida que los valores «crecen» y, posiblemente, se vaya liberando para otros usos a medida que los valores «encogen». Por ello, está claro que la representación de las estructuras avanzadas es un problema sutil y difícil cuya solución influye de forma crucial en la eficiencia de un proceso y en la economía conseguida en el uso de la memoria. Una elección adecuada sólo puede llevarse a cabo sobre la base de un conocimiento de las operaciones elementales que se vayan a realizar sobre la estructura y de su frecuencia de ejecución. Como esta información es desconocida por el diseñador del lenguaje y de su compilador es razonable que las estructuras avanzadas se encuentren excluidas de los lenguajes de uso general. Por la misma razón, los programadores deberían evitar su utilización siempre que su problema pueda tratarse mediante el empleo único de estructuras fundamentales.

En la mayor parte de los lenguajes y compiladores, se resuelve el dilema de tener que proporcionar dispositivos de estructuración avanzada de datos, sin información sobre su uso potencial, teniendo en cuenta que todas las estructuras avanzadas de datos se componen, bien de elementos no estructurados, o de estructuras fundamentales de datos. Cualquier estructura puede ser generada mediante operaciones explícitas especificadas por el programador, siempre que éste cuente con posibilidades de asignación dinámica de sus componentes, y de encadenamiento y referencia dinámicos de estos componentes. En el Capítulo 4 se tratan técnicas de generación y manipulación de estas estructuras avanzadas.

Existe, sin embargo, una estructura que puede considerarse avanzada, ya que su cardinalidad es infinita, pero cuyo uso es tan general y frecuente, que resulta casi obligada su inclusión entre las estructuras básicas: la *secuencia*. Para definir de forma abstracta el concepto de secuencia, se introduce la siguiente notación:

1. $\langle \rangle$ designa la secuencia vacía.
2. $\langle x_0 \rangle$ designa la secuencia de un único componente, x_0 ; se denomina secuencia *unitaria*.
3. Si $x = \langle x_1, \dots, x_m \rangle$ e $y = \langle y_1, \dots, y_n \rangle$ son secuencias

$$x \& y = \langle x_1, \dots, x_m, y_1, \dots, y_n \rangle \quad (1.41)$$

es la *concatenación* de x e y .

4. Si $x = \langle x_1, \dots, x_n \rangle$ es una secuencia no vacía,

$$\text{primero}(x) = x_1 \quad (1.42)$$

designa el primer elemento de x .

5. Si $x = \langle x_1, \dots, x_n \rangle$ es una secuencia no vacía

$$\text{resto}(x) = \langle x_2, \dots, x_n \rangle \quad (1.43)$$

es la secuencia x sin su primer componente. Por tanto, se verifica la relación invariante

$$\langle \text{primero}(x) \rangle \& \langle \text{resto}(x) \rangle \equiv x \quad (1.44)$$

La introducción de estas notaciones no significa que se vayan a utilizar en programas reales para ser ejecutados en computadores reales. De hecho, es esencial que la operación de concatenación *no* se use en forma generalizada, y que la manipulación de secuencias se limite a la aplicación de un conjunto de operadores cuidadosamente seleccionados, que aseguran una cierta disciplina de utilización, pero que, a su vez, se definen a partir de las nociones abstractas de secuencia y concatenación. La elección cuidadosa del conjunto de operadores de secuencias capacita a los implementadores para encontrar representaciones adecuadas y eficientes de las secuencias en cualquier tipo de memoria; esto garantiza que el mecanismo asociado de asignación dinámica de memoria sea lo suficientemente simple como para que el programador pueda trabajar sin preocuparse de los detalles.

Para poner en claro que la secuencia que se va a introducir como estructura básica, está asociada con un conjunto restringido de operadores, que fundamentalmente permiten sólo el acceso estrictamente secuencial a componentes, esta estructura se denomina *fichero secuencial* o, para abreviar, simplemente *fichero*. En forma totalmente análoga a la empleada para definición de los tipos array y conjunto, un tipo fichero se define por la fórmula

$$\text{type } T = \text{file of } T_0 \quad (1.45)$$

que expresa que cualquier fichero de tipo T consta de cero o más componentes de tipo T_0 .

EJEMPLOS

$$\begin{aligned} \text{type } \text{texto} &= \text{file of char} \\ \text{type } \text{mazo} &= \text{file of tarjeta} \end{aligned}$$

Lo principal del *acceso secuencial* es que en un momento dado sólo puede accederse de forma inmediata a un único, y especificado, componente de la secuencia. Este componente corresponde a la *posición en ese momento* del mecanismo de acceso. Esta posición puede cambiarse, por los operadores de fichero, normalmente al componente siguiente o al primero de toda la secuencia. Para expresar, de una manera formal, la posición de un fichero, se considerará un fichero x

como si estuviera dividido en dos partes, una, x_I , a su izquierda y otra, x_D , a su derecha. Evidentemente la ecuación

$$x \equiv x_I \& x_D \quad (1.46)$$

expresa una relación invariante.

Una segunda, y muy importante, consecuencia del acceso secuencial es que los procesos de construcción e inspección de una secuencia son distintos y no pueden mezclarse en cualquier orden. Así, un fichero se construye añadiendo repetidamente componentes (a su extremo final) y puede, subsiguientemente, inspeccionarse secuencialmente. Por ello, es habitual considerar que un fichero puede encontrarse bien *en estado* de ser construido (escrito) o bien en estado de ser inspeccionado (leído).

La ventaja del acceso estrictamente secuencial es particularmente notoria, si los ficheros van a estar situados en dispositivos de almacenamiento intermedio, es decir, si se presentan transferencias entre distintos dispositivos. El método de acceso secuencial es el único en el que las complejidades de los mecanismos necesarios para tales transferencias pueden ser ocultadas con éxito al programador. En particular, es posible la aplicación de simples técnicas «buffering» (memorias tampón) que garantizan por sí solas el uso óptimo de los distintos recursos disponibles en equipos complejos de proceso de datos.

Existen algunos dispositivos de almacenamiento en los que el acceso secuencial es realmente el único posible. Entre ellos están evidentemente todas las clases de cintas. Pero, incluso en los tambores magnéticos y discos, cada pista constituye un dispositivo de almacenamiento que permite únicamente el acceso secuencial. El acceso estrictamente secuencial es la característica primordial de todo dispositivo móvil, y también de algunos otros no móviles.

1.11.1. Operadores elementales de ficheros

A continuación se formula la noción abstracta de acceso secuencial mediante un conjunto de *operadores elementales de fichero*, utilizables por el programador. Se definen a partir de los conceptos de secuencia y concatenación. Hay un operador para iniciar el proceso de generación del fichero, otro para iniciar la inspección, otro para añadir un componente al final de la secuencia y otro para avanzar la inspección al componente siguiente. Los dos últimos se definen de forma que incluyen una variable implícita auxiliar que representa un «buffer». Se supone que este último se asocia automáticamente a cada variable fichero x , y se le designa por x^\uparrow . Evidentemente, si x es del tipo T , x^\uparrow es del tipo base de T , T_0 .

1. Construcción de la secuencia vacía. La operación

$$\text{rewrite}(x) \quad (1.47)$$

equivale a la asignación

$$x := \langle \rangle$$

Esta operación se realiza para escribir sobre la secuencia en curso, x , e iniciar el proceso de construcción de una nueva secuencia, y corresponde al rebobinado de una cinta.

2. Extensión de una secuencia. La operación

$$\text{put}(x) \quad (1.48)$$

corresponde a la asignación

$$x := x \& \langle x^\uparrow \rangle$$

que añade el valor x^\uparrow al final de la secuencia x .

3. Iniciación de una inspección. La operación

$$\text{reset}(x) \quad (1.49)$$

equivale a las asignaciones simultáneas

$$\begin{aligned} x_I &:= \langle \rangle \\ x_D &:= \langle x \rangle \\ x^\uparrow &:= \text{primero}(x) \end{aligned}$$

Esta operación se utiliza para iniciar el proceso de lectura de una secuencia.

4. Paso al componente siguiente. La operación

$$\text{get}(x) \quad (1.50)$$

equivale a las asignaciones simultáneas

$$\begin{aligned} x_I &:= x_I \& \langle \text{primero}(x_D) \rangle \\ x_D &:= \text{resto}(x_D) \\ x^\uparrow &:= \text{primero}(\text{resto}(x_D)) \end{aligned}$$

Obsérvese que $\text{primero}(s)$ sólo está definida si $s \neq \langle \rangle$.

Los operadores *rewrite* y *reset* no dependen de la posición del fichero previa a su ejecución; vuelven a situar el fichero, en cualquier caso, en su posición origen.

Cuando se inspecciona una secuencia es necesario estar en condiciones de reconocer el final de la misma, porque, de otra forma la asignación

$$x\uparrow := \text{primero}(x_D)$$

representará una operación no definida. Alcanzar el final del fichero es evidentemente sinónimo de que la parte a la derecha, x_D , está vacía. Por ello se introduce el predicado

$$\text{eof}(x) \equiv x_D = \langle \rangle. \quad (1.51)$$

para significar que se alcanza el fin del fichero (*end of file*). La operación $\text{get}(x)$, por tanto, sólo puede ejecutarse si el predicado $\text{eof}(x)$ es falso.

En principio, es posible expresar todas las operaciones sobre ficheros en función de cuatro operadores básicos. En la práctica, sin embargo, es frecuente combinar la operación de avanzar la posición del fichero (*get* o *put*) con el acceso a la variable buffer. Por ello, se proponen dos nuevos procedimientos; ambos pueden expresarse en función de los operadores básicos. Sea v una variable y e una expresión de tipo T_0 , componente del fichero. De acuerdo con esto

$$\text{read}(x, v) \text{ es sinónimo de } v := x\uparrow; \text{get}(x)$$

y

$$\text{write}(x, e) \text{ es sinónimo de } x\uparrow := e; \text{put}(x)$$

La ventaja de utilizar *read* y *write* en lugar de *get* y *put* reside no sólo en su brevedad, sino también en su simplicidad conceptual. Ya que, de esta forma, es posible ignorar la existencia de la variable buffer $x\uparrow$, cuyo valor algunas veces está indefinido. La variable buffer puede, sin embargo, ser útil para cuando se quiere inspeccionar el componente siguiente sin tener que avanzar la posición del fichero.

Las condiciones previas para la ejecución de los dos procedimientos son

$\neg \text{eof}(x)$	para	$\text{read}(x, v)$
$\text{eof}(x)$	para	$\text{write}(x, e)$

Al leer, el predicado $\text{eof}(x)$ toma el valor verdadero tan pronto como se ha leído el último elemento del fichero. Estas consideraciones están elegantemente incorporadas en los dos *esquemas de programas* que se indican a continuación para la construcción y proceso secuencial de un fichero x . Las instrucciones R y S y el predicado p son parámetros adicionales de los esquemas.

Escribir el fichero x :

```
rewrite(x);
while p do
  begin R(v); write(x,v)
  end
```

(1.52)

Ler el fichero x :

```
reset(x);
while  $\neg \text{eof}(x)$  do
  begin read(x,v); S(v)
  end
```

(1.53)

1.11.2. Ficheros con subestructura

En la mayoría de las aplicaciones, los grandes ficheros requieren cierta clase de subestructura. Por ejemplo, un libro, aunque puede considerarse como una secuencia única de caracteres, se subdivide en capítulos y párrafos. El objeto de esta subestructura es proporcionar algunos puntos explícitos de referencia, algunas coordenadas para facilitar la orientación en la larga secuencia de información. Algunos de los dispositivos físicos de almacenamiento facilitan la representación de estos puntos de referencia (por ejemplo, las marcas de cinta) y son capaces de situarse en ellos con mayor velocidad que cuando se inspecciona toda la información situada entre estos puntos de referencia.

De acuerdo con la notación seguida hasta ahora, la forma natural de introducir un primer nivel de subestructura es considerar el fichero como una secuencia de componentes que a su vez son secuencias, es decir, como un fichero de ficheros. Suponiendo que los componentes últimos (o unidades) son de tipo U , las subestructuras son, por tanto, de tipo

$$T' = \text{file of } U$$

y el fichero completo es del tipo

$$T = \text{file of } T'$$

Es evidente que de esta forma los ficheros pueden construirse con particiones imbricadas hasta el nivel que se desee. En general, un tipo T_n puede definirse por la relación recursiva

$$T_i = \text{file of } T_{i-1} \quad i = 1 \dots n$$

y $T_0 = U$. Tales ficheros se denominan frecuentemente *ficheros multinivel*, y

un componente de tipo T_i se denomina *segmento* de nivel i . Un ejemplo de fichero multinivel es un libro, en el que los niveles de segmentación corresponden a capítulos, apartados, párrafos y líneas. Sin embargo, el caso más general es el fichero con un único nivel de segmentación.

El fichero segmentado de un único nivel no es idéntico en absoluto a un array de ficheros, ya que el número de segmentos es variable y el fichero sólo puede extenderse por su extremo final. Continuando dentro de la convención de notación utilizada hasta ahora, y suponiendo un fichero definido por

$x : \text{file of file of } U$

x^\uparrow designa el segmento accesible en el momento de proceso, $x^\uparrow\uparrow$ el componente unitario accesible en dicho momento. De acuerdo con esto $\text{put}(x^\uparrow)$ y $\text{get}(x^\uparrow)$ se refieren a un componente unitario y $\text{put}(x)$ y $\text{get}(x)$ designan las operaciones de añadir y avanzar al segmento siguiente.

Los ficheros segmentados están implantados prácticamente en todos los dispositivos de almacenamiento incluso en cintas magnéticas. Su segmentación no ha cambiado su característica básica de permitir solamente acceso secuencial bien a componentes unitarios o bien —por mecanismos de salto más rápido— a segmentos. Otros dispositivos de almacenamiento —notablemente los *discos* y *tambores*— contienen habitualmente un número de pistas cada una de las cuales representa propiamente un dispositivo secuencial que normalmente es demasiado corto para almacenar un fichero completo. Por ello, los ficheros en disco se extienden sobre varias pistas y tienen la información de control apropiada para asegurar la conexión entre pistas. Obviamente el punto origen de cada pista constituye una marca natural de segmento, a la que puede accederse aún más directamente que a las marcas en cualquier soporte estrictamente secuencial. Una tabla de índices almacenada en memoria principal podría utilizarse, por ejemplo, para dirigir las pistas donde comienzan los segmentos e indicar su longitud (ver Figura 1.10).

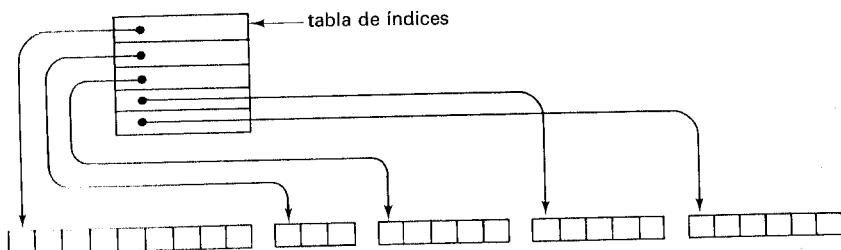


Fig. 1.10 Fichero indexado con cinco segmentos.

Las consideraciones anteriores conducen a los llamados *ficheros indexados* (llamados también algunas veces *ficheros de acceso directo*). Realmente los tam-

bones y los discos están organizados de forma que cada pista contiene muchas marcas físicas en las que puede empezar la lectura o escritura. Por ello, no es preciso que cada segmento ocupe una pista completa, ya que de esta forma resultaría una utilización muy pobre del almacenamiento disponible, si los segmentos son cortos para la longitud de pista. El tramo de almacenamiento entre marcas consecutivas se denomina *segmento físico* (o *sector*) en contraste con el *segmento lógico* que es una entidad significativa en la estructura de datos del programa. Evidentemente cada segmento físico alberga como máximo un segmento lógico, y cada segmento lógico ocupa como mínimo un segmento físico (incluso si está vacío). Debe tenerse en cuenta que aunque se denominan ficheros de «acceso directo», el tiempo medio para localizar un segmento, llamado tiempo de *latencia*, es la mitad del tiempo de una vuelta del disco.

Los ficheros indexados conservan la característica esencial de que la escritura se realiza secuencialmente a partir del último elemento. Por ello, son particularmente útiles en aplicaciones en las que se producen pocos cambios. Estos cambios se realizan, bien ampliando el fichero, o actualizándolo y volviéndolo a copiar completo. La inspección de los datos puede realizarse de forma mucho más rápida y selectiva mediante el uso de las tablas de índices. Esta situación es típica en los llamados *bancos de datos*.

Los sistemas que permiten la actualización selectiva de partes interiores del fichero son generalmente de utilización difícil y poco fiable, ya que los nuevos elementos de información deben tener el mismo tamaño que los antiguos que reemplazan. Además, en aplicaciones que afectan a grandes masas de datos, la actualización selectiva no es recomendable, habida cuenta de la regla básica de que ante cualquier fallo, ya sea debido a un programa erróneo, o al mal funcionamiento del equipo, debe existir un estado de los datos al cual pueda volverse para poder repetir el proceso que falló. Por ello, la actualización se lleva a cabo normalmente de manera completa, de forma que el fichero antiguo se sustituye por la nueva copia actualizada, sólo después de que se ha llevado a efecto una verificación que establezca que el nuevo fichero es válido. A efectos de actualización la organización secuencial es con mucho la mejor desde el punto de vista de fiabilidad. Debe preferirse a otras organizaciones más sofisticadas de grandes masas de datos, que pueden ser más eficientes, pero que, a menudo, producen la pérdida total de los datos cuando falla el equipo de proceso.

1.11.3. Textos

Los ficheros cuyos componentes son de tipo *char* juegan un papel especialmente importante en el cálculo y proceso de datos: constituyen el elemento de contacto entre los computadores y los usuarios humanos. Tanto la *entrada legible* proporcionada por los programadores, como la *salida legible* que representa los resultados calculados, están constituidos por secuencias de caracteres. Por ello, a este tipo de datos debe dársele un nombre normalizado:

```
type text = file of char
```

La comunicación entre un proceso en el computador y su autor humano se establece, finalmente, mediante un elemento puente (*interface*) que puede representarse por dos ficheros de textos. Uno de ellos contiene la *entrada* (*input*) al proceso, y el otro los resultados procesados llamados *salida* (*output*). Por ello, se supondrá la existencia de estos dos ficheros en todos los programas, siendo su declaración

```
var input, output: text
```

Teniendo en cuenta la hipótesis de que estos ficheros representan los dispositivos normalizados de un equipo de proceso (tales como la lectora de fichas y la impresora) se supondrá que el fichero *input* es sólo para lectura y el fichero *output* sólo para escritura.

Dado que ambos ficheros se utilizan predominantemente, se considerará que si el primer parámetro de los procedimientos *read* y *write* no es una variable fichero, toma el valor *input* o *output* por defecto. Avanzando en esta línea, se presentan dos procedimientos normalizados con un número arbitrario de argumentos. Los *convenios de notación* de estos procedimientos se resumen de la forma siguiente:

```
read(x1, ..., xn) equivalente a read(input, x1, ..., xn)
write(x1, ..., xn) equivalente a write(output, x1, ..., xn)
read(f, x1, ..., xn) equivalente a
begin read(f, x1); ... ; read(f, xn) end
write(f, x1, ..., xn) equivalente a
begin write(f, x1); ... ; write(f, xn) end
```

Los textos son ejemplos típicos de secuencias con subestructura. Las unidades usuales de subestructura son capítulos, párrafos y líneas. Una forma corriente de representar estas estructuras elementales en los ficheros de texto es utilizar caracteres especiales de separación. El carácter blanco es el ejemplo más conocido, pero pueden utilizarse separadores similares para marcar los finales de línea, párrafo y capítulo. Por ejemplo, el código ISO de amplia difusión (incluyendo su versión americana ASCII) contiene varios elementos de esta clase llamados *caracteres de control* (véase Apéndice A).

En este libro se ha rehusado la utilización de caracteres específicos de separación, y la definición de un método de representación de las subestructuras. En su lugar se considera un texto como fichero formado por secuencias de secuencias de caracteres, que representan las líneas. También, se limita el análisis a un nivel único de subestructura, es decir, la *línea*. Sin embargo, en vez de considerar los textos como ficheros constituidos por ficheros de caracteres imprimibles, se consideran como ficheros de caracteres, introduciendo operadores y predicados adicionales para su manipulación, es decir, para marcar y reconocer las

líneas. El efecto de ellos se comprenderá mejor si se supone que las líneas están separadas por caracteres (hipotéticos) de separación (no pertenecientes al tipo *char*) y su misión se considera que es la inserción y reconocimiento de tales separadores. Los operadores adicionales son los siguientes:

- | | |
|-------------------|--|
| <i>writeln(f)</i> | Añadir una marca de línea al final del fichero <i>f</i> . |
| <i>readln(f)</i> | Saltar caracteres en el fichero <i>f</i> hasta el siguiente inmediato a la próxima marca de línea. |
| <i>eoln(f)</i> | Función booleana. Verdadera, si la posición del fichero ha avanzado hasta una marca de línea; falsa, en otro caso (se supondrá que si <i>eoln(f)</i> es verdad <i>f</i> ↑ = blanco). |

Con esta base, es posible formular dos esquemas de programa para escribir y leer textos, semejantes a los de «escritura» y «lectura» de otros ficheros [véase (1.52) y (1.53)]. Estos esquemas se basan en un fichero de texto *f* y prestan la debida atención a la generación y reconocimiento de la estructura de línea. Sea *R(v)* una instrucción que asigna un valor a *x* (de tipo *char*) y define las condiciones *p* y *q* con significados «éste era el último carácter de la línea» y «éste era el último carácter del fichero». Sea *U* una instrucción a ejecutar al principio de cada línea leída, *S(x)* una instrucción a ejecutar para cada carácter *x* del fichero, y *V* una instrucción a ejecutar al final de cada línea.

Escritura de un texto *f*.

```
rewrite(f);
while ¬eoln(f) do
begin
  while ¬p do
    begin R(x); write(f,x)
    end ;
  writeln(f)
end
```

(1.54)

Lectura de un texto *f*.

```
reset(f);
while ¬eof(f) do
begin U;
  while ¬eoln(f) do
    begin read(f,x); S(x)
    end ;
  V; readln(f)
end
```

(1.55)

Hay casos en que la estructura de líneas en un texto no representa información relevante alguna. La hipótesis realizada sobre el valor de la variable «buffer» al encontrar una marca de línea [ver definición de *eoln(f)*] permite utilizar, en estos casos, un esquema de programa más simple. Obsérvese que de acuerdo con la definición de *eoln* cada final de línea se presenta como un carácter blanco adicional.

```
while  $\neg$  eof(f) do
  begin read(f, x); S(x)
  end
```

(1.56)

En la mayoría de los lenguajes de programación se acostumbra admitir argumentos de tipo *integer* o *real* para los procedimientos de leer y escribir. Esta generalización sería inmediata si los tipos *integer* y *real* se definieran como arrays de caracteres cuyos elementos designaran los dígitos individuales de los números. Los lenguajes fuertemente orientados hacia las aplicaciones comerciales se adaptan realmente a tales definiciones, y requieren una representación de los números a partir de los dígitos decimales de acuerdo con el sistema numérico decimal. La ventaja importante de presentar los tipos de datos *integer* y *real* como fundamentales es que estas especificaciones detalladas pueden omitirse y que un equipo de proceso puede usar diferentes representaciones de los números que sean mucho más adecuadas para sus propósitos. De hecho, los sistemas orientados hacia cálculo científico invariablemente eligen la representación binaria que en muchos aspectos es superior a la representación decimal.

Esto implica, sin embargo, que un programador no puede suponer que los números se leen o escriben en ficheros de texto sin que median operaciones de conversión. Es costumbre esconder estas operaciones de conversión en las instrucciones de lectura y escritura con argumentos de tipo numérico. El programador profesional, sin embargo, es consciente de que tales instrucciones (llamadas *instrucciones E/S o I/O*) incorporan dos funciones distintas: transferencia de datos entre diferentes dispositivos de almacenamiento y transformaciones de la representación de los datos. Estas últimas pueden ser muy complejas y costosas en tiempo.

En los capítulos siguientes de este libro, las instrucciones de lectura y escritura de argumentos numéricos se utilizarán de acuerdo con las reglas del lenguaje de programación PASCAL. Estas reglas permiten controlar el proceso de transformación, mediante un determinado tipo de especificación del formato. La especificación de formato indica el número de dígitos que se desea para el caso de instrucciones de escritura. Este número de caracteres, también llamado «amplitud de campo», se escribe inmediatamente después del argumento de la forma siguiente:

write(f, x : n)

El argumento *x* va a escribirse en el fichero *f*; su valor se convierte en una secuen-

cia (como mínimo) de *n* caracteres; si es necesario, los dígitos irán precedidos por un signo y un número adecuado de blancos.

No es preciso dar más detalles para comprender los últimos ejemplos de programas de este libro. A continuación se incluyen dos ejemplos de conversión de la representación de un número (Programas 1.3 y 1.4), con el propósito de poner de relieve la costosa *complejidad* de tales operaciones, que normalmente están implícitas en las instrucciones de escritura. Estos dos procedimientos representan la conversión de números reales desde la representación decimal a una representación «interna» arbitraria, y viceversa. (Las constantes en las cabeceras se determinan por las propiedades del formato de números en coma flotante del computador CDC 6000: exponente binario de 11 bits y mantisa de 48 bits. La función *expo(x)* designa el exponente de *x*.)

Programa 1.3 Lectura de un número real.

```
procedure leerreal (var f: texto; var x: real);
  {lee un numero real x del fichero f}
  {las constantes siguientes dependen del sistema}
const t48 = 281474976710656;   {= 2**48}
  limit = 56294995342131;   {= t48 div 5}
  z = 27;      {= ord('0')}
  lim 1 = 322;    {maximo exponente}
  lim 2 = -292;   {minimo exponente}
type posint = 0 .. 323;
var ch: char; y: real; a, i, e: integer;
  s, ss: boolean;   {signos}
function ten(e: posint): real; {= 10**e, 0 < e < 322}
  var i: integer; t: real;
begin i := 0; t := 1.0;
repeat if odd(e) then
  case i of
    0 : t := t * 1.0E1;
    1 : t := t * 1.0E2;
    2 : t := t * 1.0E4;
    3 : t := t * 1.0E8;
    4 : t := t * 1.0E16;
    5 : t := t * 1.0E32;
    6 : t := t * 1.0E64;
    7 : t := t * 1.0E128;
    8 : t := t * 1.0E256
  end;
  e := e div 2; i := i + 1
until e = 0;
ten := t
end;
```

```

begin
  {saltar los blancos iniciales}
  while f↑ = ' ' do get(f);
  ch := f↑;
  if ch = '-' then
    begin s := true; get(f); ch := f↑
    end else
    begin s := false;
      if ch = '+' then
        begin get(f); ch := f↑
        end
      end;
  end;
  if ¬(ch in ['0' .. '9']) then
    begin mensaje ('DIGITO ESPERADO'); halt;
  end;
  a := 0; e := 0;
repeat if a < limit then a := 10 * a + ord(ch) - z else e := e + 1
  get(f); ch := f↑;
until ¬(ch in ['0' .. '9']);
if ch = '.' then
begin { leer fraccion } get(f); ch := f↑;
  while ch in ['0' .. '9'] do
    begin if a < limit then
      begin a := 10 * a + ord(ch) - z; e := e - 1
      end;
      get(f); ch := f↑
    end
  end;
end;
if ch = 'E' then
begin { leer factor de escala} get(f); ch := f↑;
  i := 0;
  if ch = '-' then
    begin ss := true; get(f); ch := f↑
  end else
    begin ss := false; if ch = '+' then
      begin get(f); ch := f↑
      end
    end;
  end;
  while ch in ['0' .. '9'] do
    begin if i < limit then begin i := 10 * i + ord(ch) - z end;
      get(f); ch := f↑
    end;
  end;
  if ss then e := -e - i else e := e + i
end;

```

Programa 1.3 (Continuación)

```

if e < lim 2 then
  begin a := 0; e := 0
  end else
  if e > lim 1 then
    begin message ('NUMERO DEMASIADO GRANDE'); halt end;
  { 0 < a < 2 ** 49 }
  if a ≥ t48 then y := ((a + 1) div 2) * 2.0 else y := a;
  if s then y := -y;
  if e < 0 then x := y/ten(-e) else
  if e ≠ 0 then x := y * ten(e) else x := y;
  while (f↑ = ' ') ∧ (¬eof(f)) do get(f);
end {leerreal}

```

Programa 1.3 (Continuación)

Programa 1.4 Escribir un número real.

```

procedure escribirreal (var f: text; x: real; n: integer);
{escribir un numero real x con n caracteres en coma flotante decimal}
{las siguientes constantes dependen de las condiciones de la representacion
de numeros reales en coma flotante}
const t48 = 281474976710656; {= 2 ** 48; 48 = tamaño de la mantisa}
z = 27; {ord('0')}
type posint = 0 .. 323; {rango del exponente decimal}
var c, d, e, e0, e1, e2, i: integer;
function ten(e: posint): real; { 10 ** e, 0 < e < 322 }
  var i: integer; t: real;
begin i := 0; t := 1.0;
repeat if odd(e) then
  case i of
    0: t := t * 1.0E1;
    1: t := t * 1.0E2;
    2: t := t * 1.0E4;
    3: t := t * 1.0E8;
    4: t := t * 1.0E16;
    5: t := t * 1.0E32;
    6: t := t * 1.0E64;
    7: t := t * 1.0E128;
    8: t := t * 1.0E256
  end;
  e := e div 2; i := i + 1
until e = 0;
ten := t
end { ten };

```

```

begin { al menos se necesitan 10 caracteres: b + 9.9E + 999}
  if x == 0 then
    begin repeat write(f, ' '); n := n - 1
      until n ≤ 1;
      write(f, '0')
    end else
    begin
      if n ≤ 10 then n := 3 else n := n - 7;
      repeat write(f, ' '); n := n - 1
      until n ≤ 15;
      { 1 < n ≤ 15, numero de digitos a imprimir }
      begin { test de signo, a continuacion se obtiene el exponente}
        if x < 0 then
          begin write(f, '-'); x := -x
          end else write(f, '+');
        e := expo(x); {e = entier(log2(abs(x)))}
        if e ≥ 0 then
          begin e := e * 77 div 256 + 1; x := x/ten(e);
          if x ≥ 1.0 then
            begin x := x/10.0; e := e + 1
            end
          end else
          begin e := (e + 1) * 77 div 256; x := ten(-e) * x;
          if x < 0.1 then
            begin x := 10.0 * x; e := e - 1
            end
          end;
        {0.1 ≤ x < 1.0}
        case n of { redondeo }
          2 : x := x + 0.5E - 2;
          3 : x := x + 0.5E - 3;
          4 : x := x + 0.5E - 4;
          5 : x := x + 0.5E - 5;
          6 : x := x + 0.5E - 6;
          7 : x := x + 0.5E - 7;
          8 : x := x + 0.5E - 8;
          9 : x := x + 0.5E - 9;
          10 : x := x + 0.5E - 10;
          11 : x := x + 0.5E - 11;
          12 : x := x + 0.5E - 12;
          13 : x := x + 0.5E - 13;
          14 : x := x + 0.5E - 14;
          15 : x := x + 0.5E - 15
        end;
      end;
    end;
  end;

```

Programa 1.4 (Continuación)

```

if x ≥ 1.0 then
  begin x := x * 0.1; e := e + 1;
  end;
c := trunc(x, 48); {= trunc(x *(2 **48))}
c := 10 * c; d := c div t48;
write(f, chr(d + z), '.');
for i := 2 to n do
  begin c := (c - d * t48) * 10; d := c div t48;
  write(f, chr(d + z))
  end;
write(f, 'E'); e := e - 1;
if e < 0 then
  begin write(f, '-'); e := -e
  end else write(f, '+');
e1 := e * 205 div 2048; e2 := e - 10 * e1;
e0 := e1 * 205 div 2048; e1 := e1 - 10 * e0;
write(f, chr(e0 + z), chr(e1 + z), chr(e2 + z))
end
end {escribirreal}

```

Programa 1.4 (Continuación)

1.11.4. Un programa para edición de ficheros

Como ejemplo de aplicación de estructuras secuenciales se plantea el siguiente problema que, además, sirve para mostrar un método de desarrollo y explicación de programas. Este método se llama de *refinamiento gradual* [1.4, 1.6] y se utilizará para explicar muchos algoritmos a lo largo de este libro.

El problema es desarrollar un programa que edite un texto *x* en la forma de un texto *y*. Editar se utiliza aquí en el sentido de borrar o reemplazar determinadas líneas o insertar otras nuevas. La edición está dirigida por una secuencia de *instrucciones de edición* representadas por un texto normalizado llamado *input* de la forma siguiente:

- I, m. Inserción de un texto después de la línea *m*.
- B, *m*, *n*. Borrado de las líneas *m* hasta *n*.
- R, *m*, *n*. Reemplazamiento de las líneas *m* a *n*.
- F. Fin del proceso de edición.

Cada instrucción se escribe como una línea en el fichero normalizado *input*, al que se denomina fichero de instrucciones. *m* y *n* son números de línea, decimales,

y los textos a insertar deben seguir inmediatamente a las instrucciones *I*, *R*. La lista de instrucciones se termina por una línea en blanco.

Se supone que las instrucciones de edición se producen para números de línea *estrictamente* crecientes. Esta regla sugiere, de forma inmediata, un proceso *secuencial* del texto de entrada *x*. Está claro que el estado del proceso debe caracterizarse por la posición en cada momento del texto *x*, definida por el número de la línea que se está considerando en cada instante.

Supóngase ahora que el programa de edición se va a utilizar de manera interactiva y que, por ello, el fichero de instrucciones representa, por ejemplo, los datos que se originan en la consola de un terminal. Con esta forma de operación es muy interesante que el operador reciba alguna clase de información de comprobación. Una información útil y apropiada es el texto de la línea en la que el proceso se situó como consecuencia de la última instrucción. Esta línea se denominará *línea en curso*. Una consecuencia importante de esto último es que esta línea debe representarse explícitamente por una variable, en la que se almacena transitoriamente después de leerla de *x*, y antes de escribirla en *y*. Esta técnica se denomina de «mirada adelante». El programa de edición puede formularse de la forma siguiente:

```
program editor (x, y, input, output);
var lno: integer; {numero de la linea en curso}
    lc : linea; {linea en curso}
    x, y: text;
begin leer instrucion;
repeat interpretar instrucion;
    escribir linea;
    leer instrucion
until instrucion = 'F'
end
```

(1.57)

A continuación se procede a especificar con más detalles varias de las instrucciones. Al refinar *leer instrucion* e *interpretar instrucion* se observa que las instrucciones de edición planteadas constan de tres partes: el código de instrucción y dos parámetros. Por ello, se introducen las tres variables *codigo*, *m* y *n* para comunicación entre ambas rutinas.

```
var codigo, ch: char;
m, n: integer
```

Leer instrucción:

```
read(codigo, ch);
if ch = ',' then read(m, ch) else m := lno;
if ch = ';' then read(n) else n := m;
```

(1.58)

Esta formulación capacita para aceptar instrucciones con 0, 1 ó 2 parámetros sustituyendo apropiadamente por valores por defecto los no especificados.

Interpretar instrucción:

```
copiar;
if codigo = 'I' then
begin ponerlinea;
    insertar;
end else
if codigo = 'B' then saltar else
if codigo = 'R' then
begin insertar;
    saltar;
end else
if codigo = 'F' then copiarresto else Error
```

(1.59)

En una segunda etapa de refinamiento se expresan las instrucciones, *copiar*, *insertar* y *saltar* utilizadas en (1.59), en función de operaciones que afecten exclusivamente a líneas, es decir, en función de *traerlinea* y *ponerlinea*. Su característica común es la estructura repetitiva. *Copiar* sirve para copiar líneas de *x* a *y*, empezando por la línea en curso y terminando en la línea *m*. *Saltar* lee líneas de *x*, sin copiarlas, hasta la línea *n*.

<i>Copiar:</i> while lno < m do begin ponerlinea; traerlinea; end	while lno < m do traerlinea leerlinea; while nofin do begin ponerlinea; traerlinea end ;
<i>Saltar:</i> while lno < n do traerlinea	<i>Insertar:</i> while nofin do begin ponerlinea; traerlinea end ;
	<i>traerlinea;</i> <i>Copiarresto:</i> while \neg eof(x) do begin ponerlinea; traerlinea end ;
	<i>ponerlinea</i>

(1.60)

En la tercera y última etapa de refinamiento se expresan las operaciones *traerlinea*, *ponerlinea*, *leerlinea* y *escribirlinea* en función de operaciones con caracteres aislados. Obsérvese que, hasta ahora, todas las operaciones trabajan exclusivamente con líneas enteras y no se ha realizado ninguna hipótesis específica sobre los detalles de estructura interna de la línea. Se sabe que las líneas a su vez,

son secuencias de caracteres. Sería tentador declarar la variable *lc* (línea en curso) como una secuencia

```
var lc: file of char
```

Sin embargo, teniendo en cuenta la recomendación de que no debe usarse una estructura con cardinalidad infinita si es adecuada a los mismos fines una estructura fundamental (como el array) sería aconsejable utilizar una estructura de array en este caso. Esto es factible si se limita la longitud de línea, por ejemplo, a 80 caracteres. De acuerdo con esto se especifica

```
var lc: array [1 .. 80] of char
```

Las cuatro rutinas utilizan una variable índice *i* para este array que, en realidad, se utiliza localmente y podría muy bien declararse de esta forma en cada rutina; además, ahora es necesario introducir una variable global *L* para designar la longitud de la línea en curso.

<i>Traerlinea:</i> <code>i := 0; lno := lno + 1; while \neg eoln(<i>x</i>) do begin <i>i</i> := <i>i</i> + 1; read (<i>x</i>, <i>lc</i>[<i>i</i>]) end; <i>L</i> := <i>i</i>; readln(<i>x</i>) </code>	<i>Ponerlinea:</i> <code><i>i</i> := 0; while <i>i</i> < <i>L</i> do begin <i>i</i> := <i>i</i> + 1; write (<i>y</i>, <i>lc</i>[<i>i</i>]) end; writeln(<i>y</i>) </code>	<i>Leerlinea:</i> <code><i>i</i> := 0; while \neg eoln(<i>input</i>) do begin <i>i</i> := <i>i</i> + 1; read(<i>lc</i>[<i>i</i>]) end; readln </code>	<i>Escribirlinea:</i> <code><i>i</i> := 0; write (lno); while <i>i</i> < <i>L</i> do begin <i>i</i> := <i>i</i> + 1; write(<i>lc</i>[<i>i</i>]) end; writeln </code>
--	---	---	--

(1.61)

La condición *nofin* en la rutina *insertar* se expresa ahora en forma adecuada como

L / 0

Con esto concluye el desarrollo de este programa de edición de ficheros.

E J E R C I C I O S

- 1.1. Supóngase que los cardinales de los tipos estándares *integer*, *real* y *char* se denominan por *c_I*, *c_R*, *c_C*. ¿Cuáles son los cardinales de los siguientes tipos de datos definidos como ejemplos en este capítulo: *Sexo*, *boolean*, *diasemana*, *letra*, *digito*, *oficial*, *fila*, *alfa*, *complejo*, *persona*, *coordenada*, *charset*, *fecha*, *estadocinta*?
- 1.2. ¿Cómo se representarían las variables de los tipos listados en el Ejercicio 1.1:
 - a) en la memoria del computador?
 - b) en FORTRAN?
 - c) en otro lenguaje de programación a elegir por el lector?
- 1.3. ¿Cuáles son las secuencias de instrucciones (en el computador que decida el lector) para las siguientes operaciones:
 - a) acceso y almacenamiento a los elementos de arrays y registros empaquetados?
 - b) operaciones con conjuntos, incluido el test de pertenencia?
- 1.4. ¿Puede comprobarse el uso correcto de los registros variantes al ejecutar? ¿Puede incluso comprobarse al compilar?
- 1.5. ¿Cuáles son las razones para definir determinados conjuntos de datos como ficheros secuenciales en vez de como arrays?
- 1.6. Se quiere implementar los ficheros secuenciales de la forma definida en 1.11. en un computador con memoria muy amplia. Se permite imponer la restricción de que los ficheros no excedan una longitud determinada *L*. Por tanto, los ficheros pueden representarse como arrays.
Describir una posible implementación, que incluya la forma elegida de representación de los datos y los procedimientos para los operadores elementales de fichero *get*, *put*, *reset* y *rewrite*, que se definen por un conjunto de axiomas en 1.11.
- 1.7. Aplicar el ejercicio 1.6 para ficheros segmentados.
- 1.8. Dado un horario de ferrocarril que lista los servicios diarios en varias líneas de una compañía, se pide encontrar una representación de estos datos en términos de arrays, registros o ficheros, que sea adecuada para consultar las horas de llegada y salida para una estación dada y en la dirección deseada del tren.
- 1.9. Se supone dado un texto *T* en forma de fichero, y dos listas de un pequeño número de palabras en forma de dos arrays *A* y *B*. Se supone que las palabras son pequeños arrays de caracteres de una longitud pequeña y fija.
Escribir un programa que transforme el texto *T* en el texto *S* reemplazando cada ocurrencia de una palabra *A_i* por su correspondiente *B_i*.
- 1.10. ¿Qué ajustes —redefinición de constantes, etc.— son necesarios para adaptar los Programas 1.3 y 1.4 al ordenador utilizable por el lector?
- 1.11. Escribir un procedimiento semejante al programa 1.4 cuya cabecera sea

```
procedure imprimirreal (var f: text; x: real; n, m: integer);
```

Se supone que transforma el valor *x* en una secuencia de *n* caracteres como mínimo (para ser añadidos al extremo final del fichero *f*). *x* se representa en forma decimal en coma fija con *m* dígitos a continuación del punto decimal. Si es preciso, el número irá precedido por un número adecuado de blancos y/o un signo.

- 1.12. Volver a escribir el editor de texto de 1.11.4 en forma de un programa completo.
- 1.13. Comparar las siguientes tres versiones de la búsqueda binaria con (1.17). ¿Cuáles de los tres programas son correctos? ¿Cuáles son más eficientes?
Se suponen las siguientes variables y una constante $N > 0$:

```
var i, j, k: integer;
a: array[1 .. N] of T;
x: T
```

Programa A:

```
i := 1; j := N;
repeat k := (i + j) div 2;
  if a[k] < x then i := k else j := k
until (a[k] = x) ∨ (i ≥ j)
```

Programa B:

```
i := 1; j := N;
repeat k := (i + j) div 2;
  if x ≤ a[k] then j := k - 1;
  if a[k] ≤ x then i := k + 1
until i > j
```

Programa C:

```
i := 1; j := N;
repeat k := (i + j) div 2;
  if x < a[k] then j := k else i := k + 1
until i ≥ j
```

Nota: Todos los programas deben terminar con $a[k] = x$ si tal elemento existe y con $a[k] \neq x$ si no existe ningún elemento con valor x .

- 1.14. Una compañía organiza una encuesta para determinar el éxito de sus productos: Sus productos son discos y cintas con éxitos musicales; los éxitos más populares deben estar incluidos en una lista de éxitos (hit parade). La población encuestada se va a dividir en cuatro categorías según sexo y edad (por ejemplo, menores o iguales a veinte años y mayores de veinte años). A cada persona se le pide que dé el nombre de cinco éxitos. Los éxitos se identifican por los números 1 a N (por ejemplo $N = 30$). Los resultados de la encuesta se representan por el fichero siguiente:

```
type exito = 1 .. N;
sexo = (masculino, femenino);
respuesta =
record apellido, nombre: alfa
  s: sexo
  edad: integer
  eleccion: array [1 .. 5] of exito
end;
var encuesta: file of respuesta
```

Por tanto, cada elemento del fichero representa un encuestado e incluye su apellido, nombre, sexo, edad y sus cinco éxitos elegidos en orden de preferencia. Este fichero es la entrada a un programa que, se supone, calcula los siguientes resultados:

1. Lista de éxitos en orden de popularidad. Para cada uno se lista el número del éxito y el número de veces que ha sido mencionado en la encuesta. Los éxitos que no han sido mencionados en la encuesta se omiten de la lista.
2. Cuatro listas distintas con los nombres y apellidos de todos los encuestados que habían mencionado en primer lugar, uno de los tres éxitos más populares en su categoría.

Las cinco listas deben ir precedidas por títulos adecuados.

REFERENCIAS

1. DAHL, O. J., DIJKSTRA, E. W. y HOARE, C. A. R., *Structured Programming* (Nueva York: Academic Press, 1972), pág. 155-65.
2. HOARE, C. A. R., «Notes on Data Structuring», en *Structured Programming*, Dahl, Dijkstra, and Hoare, pág. 83-174.
3. JENSEN, K. y WIRTH, N., «PASCAL, User Manual and Report», *Lecture Notes in Computer Science*, Vol. 18 (Berlin: Springer-Verlag, 1974).
4. WIRTH, N., «Program Development by Stepwise Refinement», *Comm. ACM*, 14, n.º 4 (1971), pág. 221-27.

- 1.5. , «The Programming Language PASCAL», *Acta Informatica*, 1, n.º 1 (1971), pág. 35-63.
- 1.6. , «On the Composition of Well-Structured Programs», *Computing Surveys*, 6, n.º 4 (1974), pág. 247-59.

2 ORDENACION

2.1. INTRODUCCION

El objetivo principal de este capítulo es aportar un conjunto extenso de ejemplos que ilustren el uso de las estructuras de datos presentadas en el capítulo anterior y mostrar cómo la estructura de los datos de base influye profundamente en los algoritmos que llevan a efecto una tarea dada. Los procesos de ordenación constituyen también un buen ejemplo de cómo una tarea puede desarrollarse mediante algoritmos muy distintos, cada uno con ciertas ventajas e inconvenientes, que deben ser valorados unos respecto de otros, según la aplicación particular de que se trate.

Generalmente, se considera *ordenar* como el proceso de reorganizar un conjunto dado de objetos en una secuencia especificada. El objetivo de este proceso es facilitar la búsqueda subsiguiente de los elementos del conjunto *ordenado*. Como tal, es una actividad fundamental que se realiza universalmente. Los objetos ordenados aparecen en las guías de teléfonos, en los ficheros del impuesto sobre la renta, los índices de libros, bibliotecas, diccionarios, almacenes y casi en todas partes en que hay objetos que deben buscarse y recuperarse. Incluso a los niños pequeños se les enseña a poner sus cosas «en orden», y se les enfrenta con ciertas formas de ordenación mucho antes de que aprendan nada de aritmética.

De aquí que la ordenación sea una actividad fundamental y relevante, particularmente en proceso de datos. ¡Qué otra cosa va a ser más fácil de ordenar que los datos! A pesar de todo, el interés de tratar los procesos de ordenación en este libro se centra en las, aún más fundamentales, técnicas utilizadas en la construcción de algoritmos. Existen pocas técnicas que no se relacionen en alguna forma con los algoritmos de ordenación. En particular, este tema es idóneo para mostrar una gran diversidad de algoritmos, todos con el mismo objetivo, muchos siendo óptimos en algún aspecto, y la mayoría de ellos presentando

certas ventajas sobre los otros. Es por ello un tema ideal para hacer patente la necesidad del análisis del funcionamiento de algoritmos. Además el caso de ordenación es muy adecuado para mostrar cómo pueden obtenerse ganancias muy significativas del rendimiento mediante el desarrollo de algoritmos sofisticados, aunque se disponga de otros métodos más sencillos.

La influencia de la estructura de datos en la elección de un algoritmo —fenómeno omnipresente— es tan profunda para los problemas de ordenación que los métodos se dividen en dos categorías: *ordenación de arrays* y *ordenación de ficheros* (secuenciales). Ambas clases se denominan a menudo ordenación *interna* y *externa*, ya que los arrays se almacenan en la memoria interna rápida, de acceso aleatorio, del computador y los ficheros se colocan convenientemente en almacenamientos «externos» más lentos basados en dispositivos mecánicos (discos y cintas). La importancia de esta distinción se hace evidente en el ejemplo de ordenación de tarjetas numeradas. Estructurar las tarjetas según un array se corresponde con extenderlas todas delante del encargado de ordenarlas de forma que cada tarjeta sea visible y accesible individualmente (ver Fig. 2.1).



Fig. 2.1. Ordenación de arrays.

En cambio, estructurar las tarjetas según un fichero implica que de cada montón solamente es visible la tarjeta superior (ver Fig. 2.2). Esta restricción tiene una influencia evidente en el método de ordenación a utilizar, pero es inevitable si el número de tarjetas a extender supera el que cabe en la mesa disponible.

Antes de comenzar, se introduce la notación y terminología que se va a utilizar a lo largo del capítulo. Se suponen dados los ítems o artículos:

$$a_1, a_2, \dots, a_n$$

Ordenar consiste en permutar estos ítems

$$a_{k_1}, a_{k_2}, \dots, a_{k_n}$$

de forma que dada una *función de ordenación* f se verifique:

$$f(a_{k_1}) \leq f(a_{k_2}) \leq \dots \leq f(a_{k_n}) \quad (2.1)$$

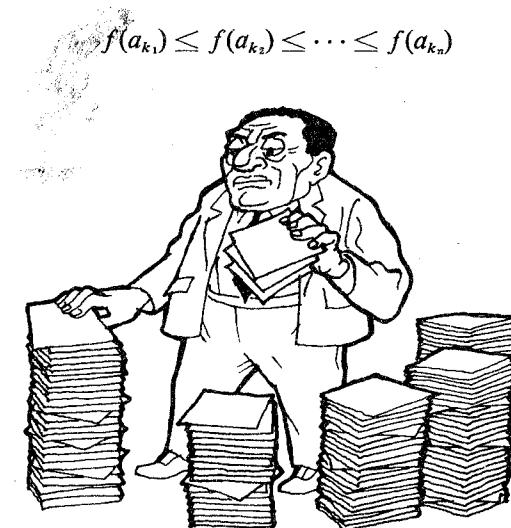


Fig. 2.2. Ordenación de ficheros.

Normalmente, la función de ordenación no se evalúa según una regla determinada de cálculo, sino que se almacena como componente explícito (campo) de cada ítem. Su valor se denomina *clave* del ítem. Por tanto, la estructura registro es particularmente idónea para representar los elementos a_i . A tal fin, se define un tipo *ítem*, que va a ser utilizado en todos los algoritmos de ordenación que siguen:

```
type item = record clave: integer;
          {aquí pueden declararse los otros componentes}
          end
```

(2.2)

Los *otros componentes*, representan los datos de interés acerca de los ítems que forman parte de la colección; la clave sirve exclusivamente para identificar cada artículo. En lo que se refiere a los algoritmos de ordenación que se van a tratar, sin embargo, la clave es el *único* componente de interés y no es preciso definir ninguno de los demás. La elección de tipo *integer* para la clave es, en cierta forma, arbitraria. Evidentemente, puede utilizarse también cualquier tipo en el que pueda definirse una relación de orden total.

Un método de ordenación se denomina *estable* si el orden relativo de los artículos con la misma clave no se altera por el proceso de ordenación. La estabi-

lidad es deseable, a veces si los ítems están ya ordenados de acuerdo con otras claves secundarias, es decir, con propiedades no reflejadas por la propia clave primaria.

Este capítulo *no debe* contemplarse como un tratado exhaustivo de las técnicas de ordenación. Mas bien, se presentan en él ejemplos muy detallados de algunas técnicas seleccionadas. Para un tratamiento más profundo del tema de ordenación, el lector interesado puede acudir al excelente compendio de D. E. Knuth [2-7] (ver también Lorin [2-10]).

2.2. ORDENACION DE ARRAYS

La principal condición a imponer a los métodos de ordenación de arrays es la utilización económica de la memoria disponible. Esto implica que las permutaciones de ítems, con vistas a su ordenación, deben realizarse utilizando el espacio ocupado por el array (ordenación *in situ*) y que los métodos que transportan artículos de un array *a* hacia un array resultado *b* son intrínsecamente de menor interés. Teniendo restringida la elección de métodos entre las muchas soluciones posibles por el criterio de economía de memoria, se clasifican a continuación éstos, según su eficiencia, es decir, su economía de tiempo. Una buena medida de eficiencia se obtiene en base al número *C* de comparaciones de claves y *M* de movimientos (transferencias) de ítems. Estos números son función del número *n* de artículos a ordenar. Aunque los buenos algoritmos de ordenación necesitan del orden de $n \cdot \log n$ comparaciones, se estudian inicialmente varias técnicas sencillas de ordenación denominadas *métodos directos*, que requieren del orden de n^2 comparaciones de claves. Hay tres buenas razones para presentar los métodos directos antes que otros algoritmos más rápidos.

1. Los métodos directos son especialmente adecuados para poner de relieve las características de los principios de ordenación más usuales.
2. Sus programas son cortos y fáciles de entender. Hay que recordar que también los programas ocupan memoria.
3. Aunque los métodos sofisticados requieren menos operaciones, éstas normalmente son más complejas en detalle; consecuentemente los métodos directos son más rápidos para *n* suficientemente pequeño, aunque no deben usarse para valores grandes de *n*.

Los métodos de ordenación *in situ*, pueden clasificarse en tres categorías principales de acuerdo con el método de base que utilizan:

1. Ordenación por inserción.
2. Ordenación por selección.
3. Ordenación por intercambio.

Estos tres principios básicos serán objeto de examen y comparación en lo que sigue.

Los programas operan sobre la variable *a*, cuyos componentes se ordenarán *in situ*, y se referirán a los tipos de datos *item* (2.2) e *indice*, definidos de la forma

```
type indice = 0 .. n;
var a: array [1 .. n] of item
```

(2.3)

2.2.1. Ordenación por inserción directa

Este método se usa generalmente por los jugadores de cartas. Los ítems (cartas) están divididos conceptualmente en una secuencia destino a_1, \dots, a_{i-1} y una secuencia origen a_i, \dots, a_n . En cada paso, empezando con $i = 2$, e incrementando i de uno en uno, se toma el elemento i de la secuencia origen y se transfiere a la secuencia destino *insertándolo* en el sitio apropiado.

Claves iniciales	44	55	12	42	94	18	06	67
$i = 2$	44	55	12	42	94	18	06	67
$i = 3$	12	44	55	42	94	18	06	67
$i = 4$	12	42	44	55	94	18	06	67
$i = 5$	12	42	44	55	94	18	06	67
$i = 6$	12	18	42	44	55	94	06	67
$i = 7$	06	12	18	42	44	55	94	67
$i = 8$	06	12	18	42	44	55	67	94

Tabla 2.1. Proceso típico de ordenación por inserción directa.

El proceso de ordenación por inserción puede verse en un ejemplo de ocho números elegidos aleatoriamente (tabla 2.1). El algoritmo es

```
for i := 2 to n do
begin x := a[i];
  «insertar x en el sitio adecuado en  $a_1 \dots a_i$ »
end
```

En el proceso de encontrar realmente el sitio adecuado conviene alternar comparaciones y movimientos, es decir, «dejar caer» *x*, comparándolo con el ítem inmediato precedente a_j ; como consecuencia, se inserta *x* o se mueve a_j a su derecha, y se continúa con el inmediato por la izquierda, según el mismo proceso. Se observa que hay dos condiciones distintas de terminación de este proceso de «caída»:

1. Se encuentra un ítem a_j cuya clave es menor que la de *x*.
2. Se alcanza el extremo izquierdo de la secuencia destino,

Este caso típico de una repetición con dos condiciones de terminación permite aplicar la conocida técnica del centinela. Se aplica fácilmente situando como centinela $a_0 = x$ (obsérvese que esto debe hacerse extendiendo el campo de variación del índice de a de 0 a n). El algoritmo completo se formula en el programa 2.1.

Programa 2.1. Ordenación por inserción directa.

```
procedure inserciondirecta;
  var i, j: indice; x: item;
begin
  for i := 2 to n do
    begin x := a[i]; a[0] := x; j := i - 1;
      while x .clave < a[j] .clave do
        begin a[j + 1] := a[j]; j := j - 1;
        end;
      a[j + 1] := x
    end
end
```

Análisis de la inserción directa. El número de comparaciones entre claves, C_i en el paso de caída i es, como máximo, $i - 1$ y, como mínimo, 1, y—suponiendo que todas las permutaciones de las n claves tienen igual probabilidad— como media $i/2$. El número de movimientos M_i (asignaciones de ítems) es $C_i + 2$ (incluido el centinela). Por ello, los números totales de comparaciones y movimientos son

$$\begin{aligned} C_{\min} &= n - 1 & M_{\min} &= 2(n - 1) \\ C_{\text{med}} &= \frac{1}{4}(n^2 + n - 2) & M_{\text{med}} &= \frac{1}{4}(n^2 + 9n - 10) \\ C_{\max} &= \frac{1}{2}(n^2 + n) - 1 & M_{\max} &= \frac{1}{2}(n^2 + 3n - 4) \end{aligned} \quad (2.4)$$

El número de comparaciones y movimientos mínimo se presenta cuando los ítems están ordenados de origen, el máximo se presenta si están inicialmente en orden inverso. En este aspecto la ordenación por inserción se comporta de forma *realmente natural*. Es evidente que el algoritmo antes descrito es un proceso estable de ordenación: no modifica el orden de los ítems con la misma clave.

El algoritmo de inserción directa se mejora fácilmente si se observa que la secuencia de destino a_1, \dots, a_{i-1} en la que debe insertarse el nuevo artículo está ya ordenada. A tal fin puede utilizarse un método más rápido para determinar el lugar de inserción. Obviamente, este método es el de búsqueda binaria que arranca del elemento central de la secuencia de destino y continúa por bisección hasta encontrar el punto de inserción. El algoritmo modificado de inserción se denomina *inserción binaria* y aparece en el Programa 2.2.

Programa 2.2. Ordenación por inserción binaria.

```
procedure insercionbinaria;
  var i, j, iz, de, m; indice: x: item;
begin
  for i := 2 to n do
    begin x := a[i]; iz := 1; de := i - 1;
      while iz ≤ de do
        begin m := (iz + de) div 2;
          if x .clave < a[m] .clave then de := m - 1 else iz := m + 1
        end;
        for j := i - 1 downto iz do a[j + 1] := a[j];
        a[iz] := x
      end
    end
end
```

Análisis de la inserción binaria. Se encuentra la posición de inserción cuando se verifica $a_{de} \cdot \text{clave} \leq x \cdot \text{clave} < a_{iz} \cdot \text{clave}$. Por tanto, el intervalo de búsqueda final debe valer 1; esto representa dividir el intervalo formado por i elementos $\lceil \log_2 i \rceil$ veces. Por tanto,

$$C = \sum_{i=1}^n \lceil \log_2 i \rceil$$

Esta sumatoria puede aproximarse por la integral

$$\int_1^n \log x \, dx = x(\log x - c) \Big|_1^n = n(\log n - c) + c \quad (2.5)$$

siendo $c = \log e = 1/\ln 2 = 1.44269\dots$ El número de comparaciones es independiente del orden inicial de los ítems. Sin embargo, debido a que la división utilizada en la etapa de bisección trunca el resultado, el número real de comparaciones necesarias con i ítems puede aumentar en una sobre las esperadas. Esta circunstancia hace que las posiciones inferiores de inserción se localicen ligeramente más rápido que las superiores. Esto favorece, por tanto, los casos en que los ítems están inicialmente muy desordenados. En efecto, se necesita el mínimo de comparaciones cuando los artículos están en orden inverso inicialmente y el máximo cuando ya están ordenados. De aquí que éste sea un caso de *comportamiento antinatural* de un algoritmo de ordenación.

$$C \approx n(\log n - \log e \pm 0.5)$$

Desgraciadamente, la mejora obtenida al utilizar la búsqueda binaria sólo se

III. ORDENACION

refleja en el número de comparaciones y no en el número de transferencias necesarias. De hecho, dado que el movimiento de los ítems, es decir, claves e información asociada es, en general, considerablemente más costoso en tiempo que la comparación de claves, esta mejora no es ni mucho menos drástica; el importante sumando M sigue siendo del orden de n^2 y, ciertamente, la reordenación de arrays ya ordenados necesita más tiempo que con la inserción directa con búsqueda secuencial. Este ejemplo demuestra que una «mejora obvia» a menudo tiene consecuencias mucho menos drásticas de lo que a primera vista puede pensarse y que, en algunos casos (que realmente se presentan), la «mejora» puede en la realidad convertirse en un empeoramiento. Después de todo, la ordenación por inserción no parece ser un método muy adecuado para los computadores: no es económica la inserción de un artículo a costa de desplazar una posición todos los elementos de una fila. Podrían esperarse mejores resultados de un método en que los movimientos de artículos se realizaran sobre ítems aislados y con saltos más largos entre posiciones de ítems. Esta idea conduce a la ordenación por selección.

2.2.2. Ordenación por selección directa

Este método se basa en los siguientes principios:

1. Seleccionar el artículo con clave mínima.
2. Intercambiarlo con el primero, a_1 .

A continuación se repiten estas operaciones con los ítems $n - 1, n - 2$, etcétera restantes hasta que quede un único artículo, el mayor. El método se ilustra con las mismas ocho claves de la tabla 2.1, en la 2.2.

Claves iniciales	44	55	12	42	94	18	06	67
	06	55	12	42	94	18	44	67
	06	12	55	42	94	18	44	67
	06	12	18	42	94	55	44	67
	06	12	18	42	94	55	44	67
	06	12	18	42	44	55	94	67
	06	12	18	42	44	55	94	67
	06	12	18	42	44	55	67	94

Tabla 2.2. Ejemplo de proceso de ordenación por selección directa.

El programa se formula de la manera siguiente:

```
for i := 1 to n - 1 do
begin «asignar a k el índice correspondiente al ítem con clave
mínima de a[1] . . . a[n]»;
«intercambiar ai y ak»
end
```

Este método, denominado de *selección directa*, es en cierta forma opuesto al de inserción directa: este considera en cada paso solamente un *único* artículo de la *secuencia origen* y *todos* los del *array de destino* para encontrar el punto de inserción; el método de selección directa considera *todos* los ítems del *array origen* para encontrar el que tiene la menor clave y depositarlo como ítem siguiente de la *secuencia de destino*. El programa completo de selección directa aparece como Programa 2.3.

Programa 2.3. Ordenación por selección directa.

```
procedure selecciondirecta;
var i, j, k: indice; x: item;
begin for i := 1 to n - 1 do
begin k := i; x := a[i];
for j := i + 1 to n do
if a[j] . clave < x . clave then
begin k := j; x := a[j]
end;
a[k] := a[i]; a[i] := x;
end
end
```

Análisis del método de selección directa. Evidentemente, el número C de comparaciones entre claves es independiente de la ordenación inicial de éstas. En este aspecto, puede decirse que este método se comporta de forma menos natural que el de inserción directa. Se obtiene

$$C = \frac{1}{2}(n^2 - n)$$

El número M de movimientos es como mínimo

$$M_{\min} = 3(n - 1) \quad (2.6)$$

en el caso de claves ordenadas inicialmente y como máximo

$$M_{\max} = \text{trunc}\left(\frac{n^2}{4}\right) + 3(n - 1)$$

si las claves están inicialmente en orden inverso. El valor medio M_{med} es difícil de determinar a pesar de que el algoritmo es sencillo. Depende del número de

veces que k_j resulta menor que todos sus precedentes k_1, k_2, \dots, k_{j-1} al recorrer una secuencia de números k_1, \dots, k_n . Este valor, medio de las $n!$ permutaciones de n claves es

$$H_n - 1$$

en donde H_n es el *número armónico n-ésimo*

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \quad (2.7)$$

(Knuth, Vol. 1, pp. 95-99).

H_n puede expresarse en la forma

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \dots \quad (2.8)$$

siendo $\gamma = 0,577216\dots$ la constante de Euler. Para valores de n suficientemente grandes pueden ignorarse los términos fraccionarios y, por ello, el número medio de asignaciones en el paso i es aproximadamente

$$F_i = \ln i + \gamma + 1$$

De acuerdo con esto, el valor medio del número de movimientos M_{med} en un proceso de ordenación por selección, es la suma de las F_i (i variando de 1 a n).

$$M_{\text{med}} = \sum_{i=1}^n F_i = n(\gamma + 1) + \sum_{i=1}^n \ln i$$

Si, además, se aproxima la suma de los términos discretos por la integral

$$\int_1^n \ln x \, dx = x(\ln x - 1) \Big|_1^n = n \ln n - n + 1$$

se obtiene como valor aproximado

$$M_{\text{med}} = n(\ln n + \gamma) \quad (2.9)$$

Como conclusión, puede afirmarse que, en general, el algoritmo de selección directa es preferible al de inserción directa, aunque en los casos en que las claves están inicialmente ordenadas o casi ordenadas, este último puede ser algo más rápido.

2.2.3. Ordenación por intercambio directo

La clasificación de un método de ordenación, raramente puede hacerse de forma clara. Los dos métodos descritos en los apartados anteriores pueden también considerarse métodos de intercambio. En este apartado, sin embargo, se presenta un método cuya característica dominante es el intercambio entre pares de ítems. El algoritmo de intercambio directo que se describe a continuación, se basa en el principio de comparar e intercambiar pares de ítems adyacentes hasta que todos estén ordenados.

Como en los métodos anteriores de selección directa, se hacen repetidas pasadas sobre el array, moviendo en cada una el elemento de clave mínima hasta el extremo izquierdo del array. En cambio, si se mira el array como si estuviera en posición vertical en vez de horizontal y —con alguna imaginación— los ítems se consideran burbujas en un depósito de agua con «pesos» acordes con sus claves, de cada pasada sobre el array resulta la ascensión de una burbuja hasta el nivel de peso que le corresponde (ver tabla 2.3). Este método se conoce generalmente como *método de la burbuja*. Su forma más simple aparece en el Programa 2.4.

inicial	2	3	4	5	6	7	∞
44	06	06	06	06	06	06	06
55	44	12	12	12	12	12	12
12	55	44	18	18	18	18	18
42	12	55	44	42	42	42	42
94	42	18	55	44	44	44	44
18	94	42	42	55	55	55	55
06	18	94	67	67	67	67	67
67	67	67	94	94	94	94	94

Tabla 2.3. Un ejemplo de ordenación por el método de la burbuja.

Programa 2.4. Programa de ordenación por el método de la burbuja.

```

procedure burbuja;
  var i, j: indice; x: item;
begin for i := 2 to n do
  begin for j := n downto i do
    if a[j - 1] · clave > a[j] · clave then
      begin x := a[j - 1]; a[j - 1] := a[j]; a[j] := x;
      end
    end
  end {burbuja}

```

Este algoritmo admite fácilmente algunas mejoras. El ejemplo de la Tabla 2.3 muestra que las tres últimas pasadas no tienen ningún efecto en el orden de los

7.2 ORDENACION

elementos por estar éstos ya ordenados. Una técnica obvia para mejorar este algoritmo, es controlar si se ha producido algún cambio en una pasada. Es necesaria, por tanto, una última pasada sin operaciones de intercambio para determinar que el algoritmo puede acabar. Sin embargo, esta mejora puede a su vez aumentarse si se controla, no sólo el hecho de que se ha producido un cambio, sino también la posición (índice) del último intercambio. Está claro, que todos los pares de ítems adyacentes por debajo de este índice, k por ejemplo, están ordenados. Las pasadas subsiguientes pueden por ello terminarse en este índice, sin tener que llegar al límite inferior i , predeterminado. El programador avisado, observará, sin embargo, una asimetría peculiar: una sola burbuja, mal situada en el extremo «pesado» de un array, por otra parte ordenado, se situará en posición correcta en una única pasada, pero un ítem mal colocado en el extremo «ligero» se «chunde» hacia su posición correcta a un ritmo de una posición por cada pasada. Por ejemplo, el array

12 18 42 44 55 67 94 06

sería ordenado por el método de la burbuja mejorado en una única pasada, en cambio, el array

94 06 12 18 42 44 55 67

necesitaría siete pasadas. Esta asimetría antinatural sugiere una tercera mejora: la alternancia en la dirección de dos pasadas consecutivas. El algoritmo resultante se denomina adecuadamente *método de la sacudida*; su forma de funcionar se ilustra en la tabla 2.4, aplicado a las mismas ocho claves utilizadas en la tabla 2.3.

Programa 2.5. Ordenación por el método de la sacudida.

```

procedure sacudida;
  var j, k, iz, de: indice; x: item;
begin iz := 2; de := n; k := n;
repeat
  for j := de downto iz do
    if a[j - 1] .clave > a[j] .clave then
      begin x := a[j - 1]; a[j - 1] := a[j]; a[j] := x;
      k := j
      end;
    iz := k + 1;
  for j := iz to de do
    if a[j - 1] .clave > a[j] .clave then
      begin x := a[j - 1]; a[j - 1] := a[j]; a[j] := x;
      k := j
      end;
    de := k - 1
  until iz > de
end {sacudida}

```

iz = 2	3	3	4	4
de = 8	8	7	7	4
↑	↓	↑	↓	↑
44	06	06	06	06
55	44	44	12	12
12	55	12	44	18
42	12	42	18	42
94	42	55	42	44
18	94	18	55	55
06	18	67	67	67
67	67	94	94	94

Tabla 2.4. Ejemplo de aplicación del método de la sacudida.

Análisis de los métodos de la burbuja y de la sacudida. El número de comparaciones en el algoritmo de intercambio directo es

$$C = \frac{1}{2}(n^2 - n) \quad (2.10)$$

y los números mínimo, medio y máximo de movimientos (asignaciones de ítems) son

$$M_{\min} = 0, \quad M_{\text{med}} = \frac{3}{4}(n^2 - n), \quad M_{\max} = \frac{3}{2}(n^2 - n) \quad (2.11)$$

El análisis de los métodos mejorados, particularmente el de la sacudida, es complejo. El mínimo número de comparaciones es $C_{\min} = n - 1$. Para el método de la burbuja mejorado, Knuth llega a un número medio de pasadas proporcional a $n - k_1 \sqrt{n}$ y un número medio de comparaciones proporcional a $\frac{3}{2}[n^2 - n(k_2 + \ln n)]$. Pero se observa que todas las mejoras antes descritas no afectan al número de intercambios; solamente reducen el número de dobles comprobaciones redundantes. Desgraciadamente el intercambio de dos ítems es una operación generalmente más costosa que una comparación. Por tanto, las hábiles mejoras introducidas tienen un efecto mucho menos profundo de lo que intuitivamente podría esperarse.

Este análisis demuestra que la ordenación por intercambio, incluidas sus leves mejoras, es inferior a la ordenación por selección o por inserción; de hecho, el método de la burbuja tiene escasas razones que lo hagan recomendable como no sea su chocante nombre. El método de la sacudida se utiliza con ventaja en los

casos en que se sabe que los artículos están casi ordenados —lo que es raro en la práctica.

Puede demostrarse que la distancia media que cada uno de los n ítems debe recorrer en un proceso de ordenación es $n/3$ lugares. Este resultado aporta una clave para buscar métodos más efectivos. Todos los métodos de ordenación directa mueven, esencialmente, cada ítem una posición, en cada paso elemental. Por tanto, necesariamente requieren del orden de n^2 de tales pasos. Cualquier mejora debe basarse en el principio de mover los ítems una distancia mayor en cada paso.

A continuación, se presentan tres métodos mejorados, uno por cada método básico de ordenación: inserción, selección e intercambio.

2.2.4. Ordenación por inserción con incrementos decrecientes

1). L. Shell propuso en 1959 un método de inserción directa mejorado. El método aparece aplicado al ejemplo ya utilizado de ocho ítems (ver tabla 2.5). En primer lugar, se agrupan y ordenan por separado, los ítems que distan cuatro posiciones. Este proceso se denomina ordenación de cuatro en cuatro. En el ejemplo de ocho ítems cada grupo tiene dos ítems. Después de este primer paso se agrupan los ítems que distan dos posiciones y se ordenan por separado. Este proceso se denomina ordenación de dos en dos. Finalmente en la tercera pasada se ordenan los ítems de manera normal, es decir, de uno en uno.

Puede preguntarse a primera vista si la necesidad de hacer varias pasadas de ordenación cada una de las cuales afecta a todos los ítems, no introducirá más trabajo del que ahorra. Sin embargo cada pasada de ordenación sobre una lista afecta a relativamente pocos artículos o bien estos están ya bastante ordenados y comparativamente se requieren pocas reorganizaciones.

Obviamente el método produce como resultado un array ordenado, y es bastante evidente que cada pasada se beneficia de las anteriores (dado que cada ordenación de i en i combina dos grupos ordenados previamente en el proceso de $2i$ en $2i$). También es obvio que cualquier secuencia de incrementos es aceptable siempre que la última sea de uno en uno porque en el peor de los casos, la última pasada realizaría toda la ordenación. Sin embargo, no es tan evidente que el método de incrementos decrecientes produzca resultados mejores con incrementos distintos de las potencias de 2.

Por tanto, el programa se desarrolla sin fijar una secuencia específica de incrementos. Los t incrementos se denominan

$$h_1, h_2, \dots, h_t$$

con las condiciones,

$$h_t = 1, \quad h_{t+1} < h_t \quad (2.12)$$

Cada ordenación de h en h se programa como un proceso de inserción directa empleando la técnica del centinela para la condición de terminación en la búsqueda de la posición de inserción.

Es evidente que cada pasada de ordenación necesita situar su centinela propio y que el programa para determinar su posición debe ser lo más sencillo posible. El array a debe extenderse, por tanto, no solamente con un componente $a[0]$ sino con h_1 componentes, de forma que su declaración es ahora:

$a : \text{array}[-h_1 \dots n] \text{ of item}$

El algoritmo, para $t = 4$, se describe mediante el procedimiento *shell* [2-11] del Programa 2.6.

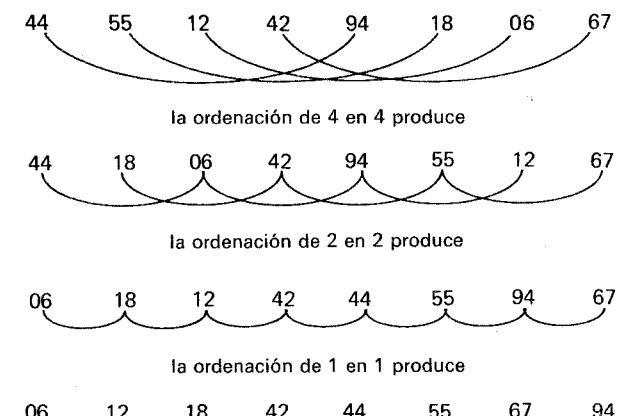


Tabla 2.5. Una ordenación por inserción con incrementos decrecientes.

Programa 2.6. Ordenación por el método de Shell.

```

procedure shell;
  const t = 4;
  var i, j, k, s: indice; x: item; m: 1 .. t;
  h: array [1 .. t] of integer;
begin h[1] := 9; h[2] := 5; h[3] := 3; h[4] := 1;
  for m := 1 to t do
    begin k := h[m]; s := -k; {posicion del centinela}
      for i := k + 1 to n do
        begin x := a[i]; j := i - k;
          if s = 0 then s := -k; s := s + 1; a[s] := x;
          while x .clave < a[j] .clave do
            begin a[j + k] := a[j]; j := j - k
            end;
            a[j + k] := x
          end;
        end;
      end;
    end;
  end

```

Análisis del método de Shell. El análisis de este algoritmo plantea algunos problemas matemáticos muy difíciles, muchos de los cuales todavía no están resueltos. En particular, no se conoce qué secuencia de incrementos produce el mejor resultado. Un hecho sorprendente es que, sin embargo, no deben ser unos múltiplos de otros. Esto evitará el fenómeno, evidente en el ejemplo presentado anteriormente, de que cada pasada combina dos cadenas que, previamente, no habían estado relacionadas. Realmente, es deseable que haya interacción lo más frecuentemente que se pueda, y se cumple el teorema siguiente:

Si una secuencia está ordenada de k en k y se ordena de i en i , continúa ordenada de k en k .

Knuth [2-8] presenta evidencias de que una elección razonable de incrementos es la secuencia (escrita en orden inverso)

$$1, 4, 13, 40, 121, \dots$$

en donde $h_{k-1} = 3h_k + 1$, $h_t = 1$ y $t = \lfloor \log_3 n \rfloor - 1$. También recomienda la secuencia

$$1, 3, 7, 15, 31, \dots$$

en donde $h_{k-1} = 2h_k + 1$, $h_t = 1$ y $t = \lfloor \log_2 n \rfloor - 1$. Para esta última, el análisis matemático evalúa el esfuerzo computacional para ordenar n elementos por el método Shell, como proporcional a $n^{1.2}$. Aunque es una mejora significativa con relación a n^2 , no se tratará con más extensión este método, ya que hay algoritmos todavía mejores.

2.2.5. Ordenación según un árbol

El método de ordenación por selección directa se basa en la repetición de la selección de la clave mínima entre n elementos, después entre los restantes $n - 1$, etcétera. Evidentemente, encontrar la clave mínima entre n ítems, necesita $n - 1$ comparaciones y encontrarla entre $n - 1$ precisa $n - 2$. ¿Cómo puede mejorarse este proceso? Solamente puede hacerse reteniendo en cada recorrido de la lista más información que la identificación del ítem mínimo. Por ejemplo, con $n/2$ comparaciones es posible determinar la clave menor de cada par de artículos, con otras $n/4$ comparaciones puede seleccionarse la menor de cada pareja de claves mínimas obtenidas anteriormente y así sucesivamente. Finalmente con sólo $n - 1$ comparaciones puede construirse un árbol de selección como el de la Figura 2.3, e identificar la raíz como la clave mínima deseada [2-2].

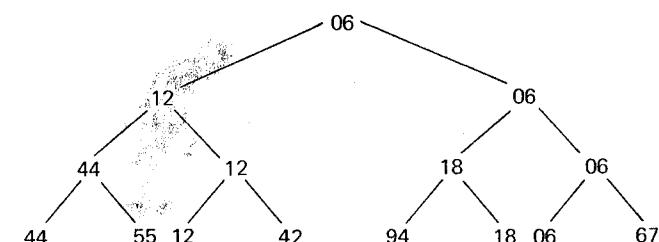


Fig. 2.3. Selección repetida entre cada dos claves.

El segundo paso consiste en descender por el camino marcado por la clave mínima, eliminándola, y reemplazándola, bien por un nudo vacío (o con clave ∞) al final del camino, o bien por el elemento en la rama alternativa en los nodos intermedios (ver Figs. 2.4 y 2.5). De nuevo el elemento que aparece en la raíz tiene la menor (realmente la segunda menor) clave y puede eliminarse de la misma forma. Después de n pasos de este tipo el árbol está vacío (es decir, todos los nodos están huecos) y el proceso de ordenación ha terminado. Obsérvese que cada uno de los pasos de selección requiere solamente $\log_2 n$ comparaciones. Por ello, el proceso total necesita solamente del orden de $n \cdot \log n$ operaciones elementales además de los n pasos necesarios para construir el árbol. Esta es una mejora muy significativa sobre los métodos directos que precisan n^2 pasos e, incluso, sobre el método de Shell que necesita $n^{1.2}$ pasos.

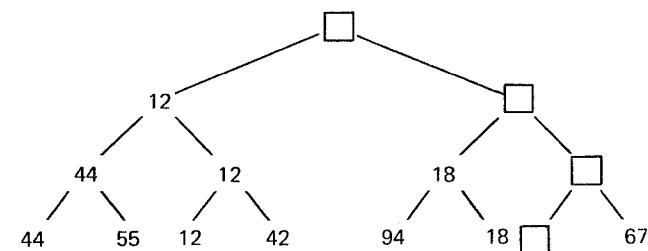


Fig. 2.4. Selección de la clave mínima.

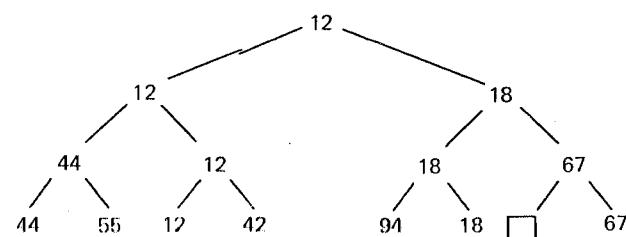


Fig. 2.5. Relleno de huecos.

Naturalmente, el manejo de datos resulta ahora más laborioso y por ello en el método de ordenación por árbol los pasos individuales son más complejos, ya que es necesario crear alguna estructura árbol para almacenar la mayor cantidad de información generada en el paso inicial. La tarea inmediata es encontrar métodos para organizar eficientemente esta información.

Desde luego sería especialmente deseable prescindir de la necesidad de los huecos ($-\infty$) que, al final, ocupan todos los nodos del árbol, y son origen de muchas comparaciones innecesarias. Además es preciso encontrar una forma de representar un árbol de n nodos en n unidades de almacenamiento, en lugar de las $2n - 1$ unidades utilizadas anteriormente. Estos objetivos se consiguen de hecho con un nuevo método, denominado del *montículo* («Heapsort») por su inventor J. Williams [2-14]; resulta evidente que este método representa una mejora drástica sobre otras formas más convencionales de ordenación con árboles.

Un *montículo* se define como una secuencia de claves

$$h_{iz}, h_{iz+1}, \dots, h_{de}$$

tales que

$$h_i \leq h_{2i} \quad (2.13)$$

$$h_i \leq h_{2i+1}$$

para todos los valores de $i = iz \dots de/2$. Si se representa un árbol binario de la forma indicada en la Fig. 2.6, se deduce que los árboles de ordenación de las Figs. 2.7 y 2.8 son *montículos* y en particular que el elemento h_1 de un *montículo* es el elemento *mínimo*.

$$h_1 = \min(h_1, \dots, h_n)$$

Supóngase dado un *montículo* con elementos $h_{iz+1} \dots h_{de}$ para valores dados iz y de , al que se quiere añadir un elemento x para formar el *montículo* extendido $h_{iz} \dots h_{de}$. Sea, por ejemplo, el *montículo* inicial que aparece en la Fig. 2.7 que se extiende «a la izquierda» con un elemento $h_1 = 44$. Se obtiene un nuevo *mon-*

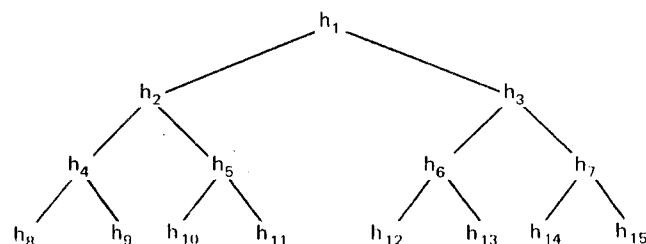


Fig. 2.6. El array h en forma de árbol binario.

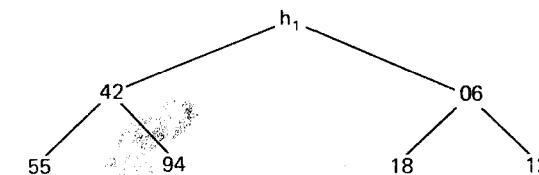


Fig. 2.7. Montículo con siete elementos.

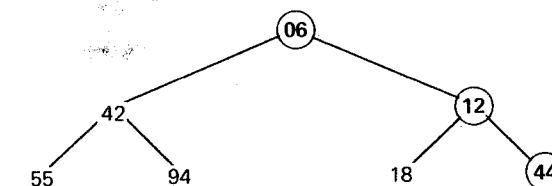


Fig. 2.8. Hundimiento de la clave 44 a través del montículo.

tículo situando inicialmente x en la cúspide del árbol y, a continuación, dejándolo «hundirse» en el *montículo*, según el camino de claves mínimas que, a su vez, se mueven hacia arriba. En el ejemplo antedicho, el valor 44 se intercambia inicialmente con 06 y a continuación con 12, formándose finalmente el árbol que aparece en la Fig. 2.8. Se comienza al lector a autoconvencerse de que el método propuesto de *criba* preserva realmente las condiciones (2.13) que definen un *montículo*.

R. W. Floyd propuso una forma elegante de construir un *montículo* sobre la propia lista de elementos. Utiliza el proceso de *criba por hundimiento* que se describe en el programa 2.7, donde i, j son los índices que designan los ítems a intercambiar en cada etapa del proceso. Se tiene un array h_1, \dots, h_n ; evidentemente los elementos $h_{n/2} \dots h_n$ ya constituyen un *montículo*, ya que ninguna pareja de índices i, j verifica que $j = 2i$ (o $j = 2i + 1$). Estos elementos forman lo que puede considerarse la fila inferior del árbol binario asociado (ver Fig. 2.6),

Programa 2.7. Criba por hundimiento:

```

procedure criba(iz, de: indice);
label 13;
var i, j: indice; x: item;
begin i := iz; j := 2 * i; x := a[i];
while j < de do
begin if j < de then
if a[j] .clave > a[j + 1] .clave then j := j + 1;
if x .clave <= a[j] .clave then goto 13;
a[i] := a[j]; i := j; j := 2 * i {criba}
end;
13: a[i] := x
end;

```

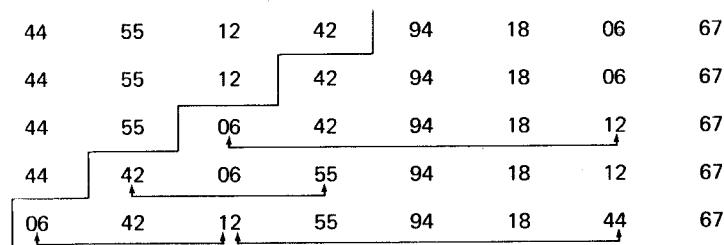


Tabla 2.6. Construcción de un montículo.

entre los que no se requiere que exista ninguna relación de orden. El *montículo* se extiende, a continuación, a la izquierda de forma que en cada paso se incluye un elemento nuevo y se sitúa adecuadamente mediante el proceso de criba por hundimiento. Este proceso se ilustra en la Tabla 2.6 y produce el *montículo* que aparece en la Fig. 2.6. Por tanto, el proceso de generar un *montículo* de n elementos $h_1 \dots h_n$ sobre la propia lista es el siguiente:

```

iz := (n div 2) + 1;
while iz > 1 do
  begin iz := iz - 1; criba(iz, n)
end
  
```

Para obtener ahora los elementos ordenados, es preciso ejecutar n pasos de criba, tomando, después de cada paso, el ítem siguiente de la cima del *montículo*. Una vez más se presenta el problema de dónde almacenar los elementos que surgen de la cima y si será posible o no una ordenación sobre la propia lista (*in situ*). ¡Desde luego que hay solución! En cada paso se saca del *montículo* el último componente (sea x), se almacena el elemento superior del *montículo* en la posición, ahora libre, de x , y se criba x por hundimiento hasta su posición correcta. Los $n - 1$ pasos necesarios se ilustran en el ejemplo de la Tabla 2.7. El proceso se describe

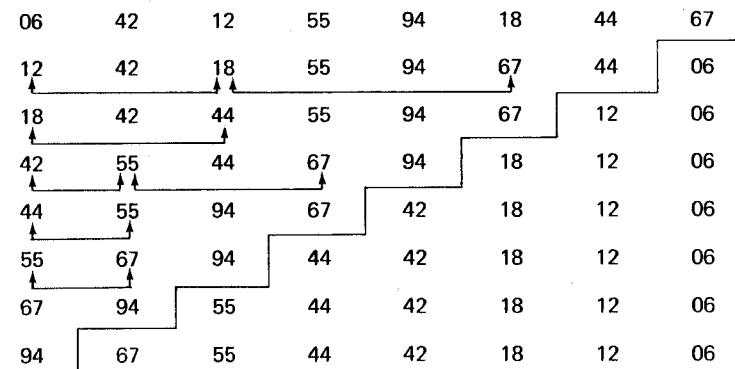


Tabla 2.7. Ejemplo de ordenación por el método del montículo.

mediante el procedimiento *criba* (Programa 2.7) de la forma siguiente

```

de := n;
while de > 1 do
  begin x := a[1]; a[1] := a[de]; a[de] := x;
        de := de - 1; criba(1, de)
  end
  
```

El ejemplo de la Tabla 2.7 muestra que el orden resultante está realmente invertido. Esto puede remediararse, sin embargo, cambiando la dirección de las relaciones de orden en el procedimiento de criba. De aquí resulta el procedimiento *montículo* que se muestra en el programa 2.8.

Programa 2.8. Ordenación por el método del montículo (heap).

```

procedure monticulo;
var iz, de: indice; x: item;
procedure criba;
label 13;
var i, j: indice;
begin i := iz; j := 2 * i; x := a[i];
while j ≤ de do
  begin if j < de then
        if a[j] .clave < a[j + 1] .clave then j := j + 1;
        if x .clave ≥ a[j] .clave then goto 13;
        a[i] := a[j]; i := j; j := 2 * i
  end;
13: a[i] := x
end;
begin iz := (n div 2) + 1; de := n;
while iz > 1 do
  begin iz := iz - 1; criba
  end;
while de > 1 do
  begin x := a[1]; a[1] := a[de]; a[de] := x;
        de := de - 1; criba
  end
end {monticulo}
  
```

Análisis del método del montículo. A primera vista no resulta evidente que este método produzca buenos resultados. Al fin y al cabo, los ítems mayores se criban primeramente hacia la izquierda para ser finalmente depositados en el extremo derecho. Realmente el procedimiento no es recomendable para un número pequeño de ítems, tal como el que se muestra en el ejemplo. Sin embargo,

II2 ORDENACION

para n grande este método es muy eficiente y su eficacia aumenta cuánto mayor es n , incluso más que la del método de Shell.

En el caso más desfavorable, se necesitan $n/2$ pasos de criba de los ítems a través de $\log(n/2)$, $\log(n/2 + 1)$, ..., $\log(n - 1)$ posiciones, siendo el logaritmo en base 2, truncado al entero más próximo, por defecto. A continuación, la fase de ordenación necesita $n - 1$ cribados con números de movimientos máximos de $\log(n - 1)$, $\log(n - 2)$, ..., 1. Además se necesitan $n - 1$ movimientos para situar el elemento cribado a la derecha. Estas consideraciones muestran que el método del *montículo* requiere del orden de $n \log n$ pasos *en el caso más desfavorable*. Este excelente rendimiento, incluso en las peores condiciones, es una de las ventajas más importantes del método.

No está claro en qué casos puede esperarse el peor (o mejor) rendimiento. Pero generalmente este método parece funcionar mejor con secuencias iniciales en las que los ítems están más o menos en orden inverso y por ello su comportamiento es antinatural. Evidentemente, la fase de creación del *montículo* no requiere ningún movimiento si los elementos están en orden inverso. Para los ocho ítems del ejemplo, las secuencias que se indican a continuación requieren el número mínimo y máximo de movimientos:

$$M_{\min} = 13 \text{ para la secuencia}$$

94 67 44 55 12 42 18 6

$$M_{\max} = 24 \text{ para la secuencia}$$

18 42 12 44 6 55 67 94

El número medio de movimientos es, aproximadamente, $\frac{1}{2}n \cdot \log n$ y las desviaciones respecto de este valor son relativamente pequeñas.

2.2.6. Ordenación por partición

Después de estudiar dos métodos avanzados de ordenación basados en los principios de inserción y selección, se presenta un tercer método mejorado basado en el principio de intercambio. Dado que el método de la burbuja era como media el menos efectivo de los tres algoritmos de ordenación directa, puede esperarse un factor de mejora relativamente importante. A pesar de todo resulta sorprendente que la mejora, basada en intercambios, que se describe a continuación produzca el mejor método de ordenar arrays conocido hasta ahora. Su rendimiento es tan espectacular que su inventor, C. A. R. Hoare, lo bautizó como *método rápido* (*Quicksort*) [2-5 y 2-6].

El método rápido se basa en el hecho de que los intercambios deben realizarse preferiblemente sobre distancias largas para que sean más efectivos. Supóngase dados n ítems en orden inverso al de sus claves. Es posible ordenarlos realizando únicamente $n/2$ intercambios, tomando en primer lugar los extremos

izquierdo y derecho y progresando gradualmente hacia el interior desde ambos lados. Naturalmente esto es posible por que se conoce que su orden inicial es exactamente el inverso del que se busca. Pero este ejemplo enseña algo, de todas formas.

Se va a intentar el siguiente algoritmo: Se toma arbitrariamente un elemento (designado por x); se inspecciona el array de izquierda a derecha hasta encontrar un ítem $a_i > x$, y entonces se inspecciona el array de derecha a izquierda hasta encontrar un ítem $a_j < x$. A continuación se intercambian los dos ítems y se continúa este proceso de inspección e intercambio hasta que los recorridos en ambas direcciones se encuentren en algún punto situado aproximadamente en la mitad del array. Como resultado se obtiene el array partido en dos; una parte izquierda con claves menores que x y una parte derecha con claves mayores que x . Este proceso de partición se formula como procedimiento en el programa 2.9. Obsérvese que las relaciones $>$ y $<$ se han sustituido por \geq y \leq cuyas negaciones en la sentencia **while** son $< y >$ respectivamente. Con este único cambio, x actúa como centinela en ambos procesos de inspección.

Programa 2.9. Partición de un array.

```
procedure particion;
var w, x: item;
begin i := 1; j := n;
  {elección aleatoria de un ítem x};
repeat
  while a[i] .clave < x .clave do i := i + 1;
  while x .clave < a[j] .clave do j := j - 1;
  if i ≤ j then
    begin w := a[i]; a[i] := a[j]; a[j] := w;
    i := i + 1; j := j - 1
    end
  until i > j
end
```

Por ejemplo, si se toma la clave media 42 como elemento x de comparación, el array de claves

44 55 12 42 94 06 18 67

necesita dos intercambios para obtener el array particionado:

↓ 18 ↓ 06 12 | 42 | 94 55 44 67

Los valores finales de los índices son $i = 5$ y $j = 3$. Las claves a_1, \dots, a_{i-1}

son menores o iguales que la clave $x = 42$, las claves $a_{j+1} \dots a_n$ son mayores o iguales que la clave x . Consiguientemente hay dos particiones, es decir

$$\begin{aligned} a_k .clave \leq x .clave & \quad \text{para } k = 1 \dots i - 1 \\ a_k .clave \geq x .clave & \quad \text{para } k = j + 1 \dots n \end{aligned} \quad (2.14)$$

y, por tanto,

$$a_k .clave = x .clave \quad \text{para } k = j + 1 \dots i - 1$$

Este algoritmo es muy directo y eficiente, ya que los elementos de comparación principales pueden mantenerse en registros rápidos a lo largo del proceso de inspección. Sin embargo, también puede ser engoroso como ocurre en el caso de n claves idénticas que requiere $n/2$ intercambios. Estos intercambios innecesarios pueden eliminarse fácilmente sustituyendo las instrucciones de inspección por

```
while  $a[i] .clave \leq x .clave$  do  $i := i + 1;$ 
while  $x .clave \leq a[j] .clave$  do  $j := j - 1;$ 
```

En este caso, en cambio, el elemento seleccionado x , que forma parte del array, ya no actúa como centinela para los dos procesos de inspección. El array con todas las claves idénticas hará que estos procesos sobrepasen los límites del array a menos que se utilicen condiciones de terminación más complicadas. La sencillez de las condiciones utilizadas en el programa 2.9 compensa de los intercambios extra que se presentan raras veces en los casos «aleatorios» corrientes. Puede obtenerse, sin embargo, un pequeño ahorro sustituyendo la cláusula de control del paso de intercambio por

$$i < j$$

en lugar de $i \leq j$. Sin embargo, esta modificación no debe hacerse extensiva a las instrucciones

$$i := i + 1; \quad j := j - 1$$

que, por tanto, requieren además una cláusula condicional propia. La necesidad de esta cláusula se pone de relieve en el siguiente ejemplo con $x = 2$:

1 1 1 2 1 1 1

La primera pasada de inspección e intercambio produce

1 1 1 1 1 1 2

con $i = 5$, $j = 6$. La segunda pasada deja el array sin modificar con $i = 7$, $j = 6$. Si el intercambio no hubiera estado controlado por la condición $i \leq j$ se habría llevado a efecto un intercambio erróneo entre a_6 y a_7 .

Puede entenderse mejor el algoritmo de partición comprobando que las dos condiciones (2.14) son invariantes en la instrucción **repeat**. Al principio se satisfacen obviamente para $i = 1$ y $j = n$ y a la salida la condición $i > j$ implica el resultado deseado.

No hay que olvidar que el objetivo perseguido no era sólo realizar particiones del array inicial, sino también ordenarlo. Sin embargo, solo hay un paso entre la partición y la ordenación: después de hacer la partición se aplica el mismo proceso a cada una de las partes y a continuación a las partes resultantes y así sucesivamente, hasta que cada partición contenga un único ítem. Este proceso se describe en el programa 2.10.

Programa 2.10. Ordenación por el método rápido.

```
procedure rapido;
  procedure sort(iz, de: indice);
    var i, j: indice; x, w: item;
    begin i := iz; j := de;
      x := a[(iz + de) div 2];
      repeat
        while  $a[i] .clave < x .clave$  do  $i := i + 1;$ 
        while  $x .clave < a[j] .clave$  do  $j := j - 1;$ 
        if  $i \leq j$  then
          begin w := a[i]; a[i] := a[j]; a[j] := w;
            i := i + 1; j := j - 1
          end
        until  $i > j$ ;
        if  $iz < j$  then sort(iz, j);
        if  $i < de$  then sort(i, de)
      end;
    begin sort(1, n)
    end {rapido}
```

El procedimiento **sort** se activa él mismo recursivamente. Esta forma de utilizar la recursividad en algoritmos es un instrumento muy poderoso que se discutirá más ampliamente en el Capítulo 3. En algunos lenguajes de programación más antiguos no existe la recursión por ciertas razones técnicas. A continuación se muestra cómo puede expresarse este mismo algoritmo como procedimiento no recursivo. Obviamente la solución es expresar la recursión de manera iterativa y para ello son necesarias determinadas operaciones adicionales de «administración» de datos.

La clave para construir una solución iterativa está en mantener una lista de particiones pendientes de ejecutar: Después de cada paso se generan dos nuevas

tareas de partición. Sólo una de ellas puede tratarse en la iteración siguiente; la otra debe añadirse a la lista de tareas pendientes. Desde luego, es fundamental que la lista de particiones pendientes se trate en un orden específico, concretamente, en orden inverso. Esto implica que la primera partición pendiente de la lista es la última a procesar y recíprocamente; la lista se procesa como pila. En la versión no recursiva del método rápido que se describe a continuación, cada tarea pendiente se representa sencillamente por los índices izquierdo y derecho que acotan el tramo a particionar posteriormente. De acuerdo con esto, se introduce una variable array denominada *pila* y un índice *p* que designa el último elemento incorporado a la misma. En el análisis subsiguiente del método rápido se estudiará la forma apropiada de elegir el tamaño de la pila *m*.

Programa 2.11. Versión no recursiva del método rápido.

```

procedure rapido1;
const m = 12;
var i, j, iz, de: indice;
x, w: item;
p: 0 .. m;
pila: array [1 .. m] of
record iz, de: indice end;
begin p := 1; pila[1].iz := 1; pila[1].de := n;
repeat {tomar la demanda superior de la pila}
  iz := pila[p].iz; de := pila[p].de; p := p - 1;
  repeat {subdivision de a[iz] .. a[de]}
    i := iz; j := de; x := a[(iz + de) div 2];
    repeat
      while a[i].clave < x.clave do i := i + 1;
      while x.clave < a[j].clave do j := j - 1;
      if i ≤ j then
        begin w := a[i]; a[i] := a[j]; a[j] := w;
          i := i + 1; j := j - 1
        end
      until i > j;
      if i < de then
        begin {almacenar en la pila la demanda para ordenar la
          partición derecha}
          p := p + 1; pila[p].iz := i; pila[p].de := de
        end;
        de := j
      until iz ≥ de
    until p = 0
end {rapido1}

```

Ánalisis del método rápido. Para analizar el funcionamiento de este método es preciso investigar, previamente, el comportamiento del proceso de partición. Una vez elegido un límite *x*, se recorre el array en su totalidad. De aquí que se lleven a cabo exactamente *n* comparaciones. El número de intercambios puede determinarse en base al siguiente razonamiento probabilístico.

Se supone que el conjunto de datos que va a ser objeto de partición consta de *n* claves $1 \dots n$ y que se ha elegido *x* como límite. Tras el proceso de partición *x* estará en la posición *x*-ésima del array. El número de intercambios necesarios será igual al número de elementos en la partición izquierda, $x - 1$, multiplicado por la probabilidad de que haya que intercambiar una clave. Una clave se cambia si no es menor que el límite *x*. Esta probabilidad es $(n - x + 1)/n$. El número esperado de intercambios se obtiene sumando para todos los límites posibles y dividiendo por *n*

$$M = \frac{1}{n} \sum_{x=1}^n \frac{n-x}{n} \cdot (n-x+1) = \frac{n}{6} - \frac{1}{6n} \quad (2.15)$$

Por tanto, el número esperado de intercambios es aproximadamente $n/6$.

En el caso más favorable, cuando se selecciona siempre la mediana como límite, cada proceso de partición divide el array en dos partes iguales y el número necesario de pasadas para ordenarlo es $\log n$. Por tanto, el total de comparaciones resultante es $n \cdot \log n$ y el número total de intercambios $n/6 \cdot \log n$. Evidentemente no puede esperarse que se acierte siempre en la mediana.

De hecho, la probabilidad de que ocurra esto es solamente $1/n$. Sorprendentemente, sin embargo, el rendimiento medio del método rápido es inferior al caso óptimo en un factor solamente de $2 \cdot \ln 2$ si se elige aleatoriamente el límite.

A pesar de ello, este método tiene limitaciones. La primera, que funciona sólo moderadamente bien para pequeños valores de *n*, como todos los métodos avanzados. Su ventaja sobre otros métodos avanzados es la facilidad con que puede incorporarse un método de ordenación directa para tratar las particiones pequeñas. Esto es particularmente ventajoso cuando se considera la versión recursiva del programa.

Todavía queda la cuestión del caso más desfavorable. ¿Cómo se comporta este método en ese caso? La respuesta, desgraciadamente, es decepcionante y revela el único punto débil de este método rápido (que en estos casos se convierte en el «lento»). Considérese, por ejemplo, el caso infeliz en que, cada vez, se toma como elemento de comparación el mayor elemento de una partición. En este caso, en cada etapa se divide un segmento de *n* ítems en una partición izquierda, con $n - 1$ elementos y otra partición derecha con un único elemento. El resultado es que se precisan *n* procesos de subdivisión en vez de $\log n$ y que el rendimiento en el caso más desfavorable es del orden de n^2 .

Aparentemente el paso crucial es la selección del elemento de comparación *x*. En el programa ejemplo se toma el elemento medio. Obsérvese que también

puede seleccionarse el primero o el último elemento $a[iz]$ o $a[de]$. En estos casos la situación más desfavorable se produce con el array ordenado ya inicialmente; en estas circunstancias el método rápido muestra claramente su desagrado por las tareas triviales y su preferencia por los arrays desordenados. Si se toma el elemento medio, esta extraña característica del método no aparece tan obvia, ya que ¡el caso de array ordenado inicialmente resulta el óptimo! De hecho, el rendimiento medio es ligeramente mejor si se elige el elemento medio. Hoare propone que la elección de x se realice «aleatoriamente» o seleccionándolo como mediana de una pequeña muestra de, por ejemplo, tres claves [2.12 y 2.13]. Esta prudente elección apenas influye en el rendimiento medio del método, pero mejora considerablemente el del caso más desfavorable. Resulta evidente que la ordenación empleando el método rápido es, en cierta forma, como un juego en el que hay que ser consciente de cuánto puede perderse si hay mala suerte.

Hay una lección importante que se desprende de esta experiencia, y que afecta directamente al programador. ¿Cuáles son las consecuencias del comportamiento en el caso más desfavorable, mencionado anteriormente, en el rendimiento del programa 2.11? Se ha comprobado que de cada subdivisión resulta una partición derecha de un elemento único; la demanda de ordenar esta partición se almacena en una pila para ulterior ejecución. Consiguientemente, el número máximo de demandas, y por tanto, el tamaño de la pila, es n . Esto, desde luego, es absolutamente inaceptable (obsérvese que no resulta mejor —de hecho es todavía peor— con la versión recursiva porque un sistema que permita la activación recursiva de procedimientos tendrá que almacenar automáticamente los valores de los parámetros y las variables locales de todas las activaciones de procedimientos y utilizará implícitamente una pila para esto). La forma de remediar este problema es almacenar la demanda de ordenación de la partición más larga, y continuar directamente con las nuevas subdivisiones de las particiones más cortas. De esta forma, el tamaño de la pila m se limita a $m = \log_2 n$.

El cambio a realizar en el programa 2.11 se localiza en la sección que construye las nuevas demandas. Ahora sería así:

```

if  $j - iz < de - i$  then
begin if  $i < de$  then
  begin {almacenar en pila la demanda de ordenacion de la particion derecha}
     $p := p + 1$ ;  $pila[p] .iz := iz$ ;  $pila[p] .de := de$ 
  end;
   $de := j$  {continuar ordenando la particion izquierda}
end else
begin if  $iz < j$  then
  begin {almacenar en pila la demanda de ordenacion de la particion izquierda}
     $p := p + 1$ ;  $pila[p] .iz := iz$ ;  $pila[p] .de := j$ 
  end;
   $iz := i$  {continuar ordenando la particion derecha}
end

```

(2.16)

2.2.7. Obtención de la mediana

La mediana de n ítems se define como aquel que es menor (o igual) que la mitad de los artículos y es mayor (o igual) que la otra mitad. Por ejemplo, la mediana de

16 12 99 95 18 87 10

es 18.

El problema de hallar la mediana se relaciona habitualmente con el de ordenación, ya que un método seguro de encontrarla es ordenar los n ítems y seleccionar el ítem situado en el centro. Sin embargo, el método de partición del Programa 2.9 proporciona un procedimiento potencialmente mucho más rápido. El método que va a presentarse puede generalizarse fácilmente para resolver el problema de hallar el elemento número k en orden de menor a mayor de n ítems. Encontrar la mediana constituye el caso concreto $k = n/2$.

El algoritmo, inventado por C. A. R. Hoare [2-4] funciona de la forma siguiente. En primer lugar se aplica la operación de partición del método rápido para $iz = 1$, $de = n$ tomando $a[k]$ como valor (límite) de subdivisión, x . Los índices resultantes i , j son tales que

1. $a[h] \leq x$ para todo $h < i$
 2. $a[h] \geq x$ para todo $h > j$
 3. $i > j$
- (2.17)

Pueden presentarse tres casos posibles.

1. El valor de x empleado para subdivisión era demasiado pequeño; consiguientemente el límite entre las dos particiones es inferior al valor deseado k . El proceso de partición debe repetirse con los elementos $a[i], \dots, a[de]$ (ver Fig. 2.9).

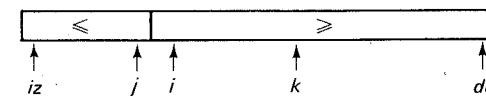


Fig. 2.9. Límite demasiado pequeño.

2. El límite elegido x era demasiado grande. Hay que repetir la operación de subdivisión con la partición $a[iz] \dots a[j]$ (ver Fig. 2.10).

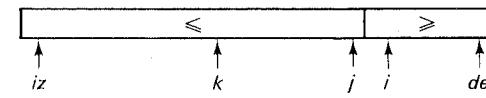


Fig. 2.10. Límite demasiado grande.

3. $j < k < i$: el elemento $a[k]$ subdivide el array en dos particiones en la proporción especificada y por tanto es el valor buscado (ver Fig. 2.11).

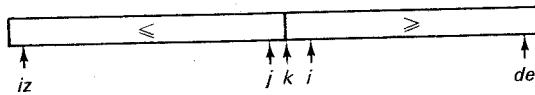


Fig. 2.11. Límite correcto.

Hay que repetir el proceso de subdivisión hasta que se presenta el caso 3. Esta iteración se expresa mediante la siguiente parte de programa:

```

 $iz := 1; de := n;$ 
while  $iz < de$  do
  begin  $x := a[iz];$ 
    particion(a[iz] . . . a[de]);
    if  $j < k$  then  $iz := i;$ 
    if  $k < i$  then  $de := j$ 
  end

```

(2.18)

El lector puede encontrar una demostración formal de este algoritmo en el artículo original de Hoare. El programa completo *encontrar* se obtiene inmediatamente

Programa 2.12. Programa de búsqueda del elemento k -ésimo.

```

procedure encontrar ( $k$ : integer);
  var  $iz, de, i, j, w, x$ : integer;
begin  $iz := 1; de := n;$ 
  while  $iz < de$  do
    begin  $x := a[iz]; i := iz; j := de;$ 
      repeat {subdivision}
        while  $a[i] < x$  do  $i := i + 1;$ 
        while  $x < a[j]$  do  $j := j - 1;$ 
        if  $i \leq j$  then
          begin  $w := a[i]; a[i] := a[j]; a[j] := w;$ 
             $i := i + 1; j := j - 1$ 
          end
      until  $i > j;$ 
      if  $j < k$  then  $iz := i;$ 
      if  $k < i$  then  $de := j$ 
    end
  end {encontrar}

```

Si se supone que, por término medio, cada subdivisión divide en partes iguales

la partición en que se encuentra el valor que se busca, el número de comparaciones necesario es:

$$n + \frac{n}{2} + \frac{n}{4} + \dots + 1 = 2n \quad (2.19)$$

es decir, es del orden de n . Esto explica la potencia del programa *encontrar* para obtener medianas y valores similares y también explica su superioridad sobre el método directo de ordenar el conjunto de ítems como paso previo a la selección del k -ésimo (que, en el mejor de los casos, es del orden de $n \cdot \log n$). En el caso más desfavorable, sin embargo, cada paso de partición reduce el tamaño del conjunto de candidatos sólo en un elemento, dando lugar a un número de comparaciones del orden de n^2 . Se insiste en que apenas hay ventaja si se utiliza este algoritmo con un número pequeño de elementos, por ejemplo, menor de 10.

2.2.8. Comparación de los métodos de ordenación de arrays

Para terminar con este desfile de métodos de ordenación, se va a intentar comparar su eficacia. Se llama n al número de ítems a ordenar y C y M siguen denominando, respectivamente, los números necesarios de comparaciones y movimientos. Pueden darse fórmulas analíticas exactas para los tres métodos de ordenación directa. Se presentan en la Tabla 2.8. Los nombres de columna min, máx, med indican, respectivamente, los valores mínimos, máximos y medios esperados, promediados con todas las $n!$ permutaciones de los n ítems.

		Min	Med	Max
Inserción directa	$C =$ $M =$	$n - 1$ $2(n - 1)$	$(n^2 + n - 2)/4$ $(n^2 - 9n - 10)/4$	$(n^2 - n)/2 - 1$ $(n^2 + 3n - 4)/2$
Selección directa	$C =$ $M =$	$(n^2 - n)/2$ $3(n - 1)$	$(n^2 - n)/2$ $n(\ln n + 0.57)$	$(n^2 - n)/2$ $n^2/4 + 3(n - 1)$
Intercambio directo (burbuja)	$C =$ $M =$	$(n^2 - n)/2$ 0	$(n^2 - n)/2$ $(n^2 - n)*0.75$	$(n^2 - n)/2$ $(n^2 - n)*1.5$

Tabla 2.8. Comparación de métodos directos de ordenación.

No se dispone de fórmulas aproximadas razonablemente simples para los métodos avanzados. Los resultados más notables son que el esfuerzo de computación necesario es $c_l \cdot n^{1.2}$ en el caso del método de Shell y $c_t \cdot \log(n)$ para los métodos rápido y del montículo. Estas fórmulas proporcionan una medida poco precisa del rendimiento como funciones de n , y permiten clasificar los algo-

ritmos de ordenación en métodos elementales o directos (n^2) y métodos avanzados o «logarítmicos» ($n \cdot \log n$). Para usos prácticos, sin embargo, es conveniente tener algunos datos experimentales que arrojen alguna luz sobre los coeficientes c_1 , a fin de clasificar con más precisión los diferentes métodos. Además, las fórmulas no tienen en cuenta el esfuerzo computacional invertido en otras operaciones aparte de las comparaciones entre claves y movimientos de ítems, tales como control de bucles, etc. Evidentemente, estos factores dependen en cierta forma del equipo concreto en que se procesa, pero, a pesar de ello, un ejemplo de datos obtenidos experimentalmente tiene interés informativo. La Tabla 2.9 muestra los tiempos (en milisegundos) consumidos por los métodos de ordenación estudiados anteriormente, ejecutados por el sistema PASCAL de un computador CDC 6400. Los tres bloques contienen los tiempos utilizados para ordenar un array que está ya inicialmente en orden, una permutación aleatoria del array y el array ordenado en sentido inverso. Las columnas izquierdas de cada bloque corresponden a 256 ítems y las derechas a 512. Los datos de la tabla separan claramente los métodos de tipo n^2 de los de tipo $n \cdot \log n$. Son de notar los siguientes puntos:

	Ordenado	Aleatorio	Orden inverso		
Inserción directa	12	23	366	1444	704
Inserción binaria	56	125	373	1327	662
Selección directa	489	1907	509	1956	695
Burbuja	540	2165	1026	4054	1492
Burbuja con señal	5	8	1104	4270	1645
Sacudida	5	9	961	3642	1619
Shell	58	116	127	349	157
Montículo	116	253	110	241	104
Rápido	31	69	60	146	37
Mezcla *	99	234	102	242	99
					232

* Véase Apartado 2.31.

Tabla 2.9. Tiempos de ejecución de programas de ordenación.

- La mejora del método de inserción binaria sobre el de inserción directa es marginal, e incluso es negativa en el caso de ya existir un orden.
- El método de la burbuja es definitivamente el peor método de ordenación. Su versión mejorada, el método de la sacudida es, incluso, peor que los de inserción directa y selección directa (excepto en el caso patológico de ordenación de un array ya ordenado).
- El método rápido supera al del *montículo* por un factor de 2 a 3. Ordena el array en orden inverso con una velocidad prácticamente idéntica a la correspondiente a un array ya ordenado.

Hay que añadir que se utilizaron datos formados únicamente por la clave, sin otros datos asociados. Esta hipótesis no es muy realista. La Tabla 2.10 muestra

cómo influye la ampliación del tamaño de los ítems. En el ejemplo elegido, los datos asociados ocupan siete veces el espacio que utiliza la clave. La columna izquierda de cada bloque presenta el tiempo necesario sin datos asociados, la columna derecha se refiere a la ordenación con datos asociados; $n = 256$.

	Ordenado	Aleatorio	Orden inverso		
Inserción directa	12	46	366	1129	704
Inserción binaria	56	76	373	1105	662
Selección directa	489	547	509	607	695
Burbuja	540	610	1026	3212	1492
Burbuja con señal	5	5	1104	3237	1645
Sacudida	5	5	961	3071	1619
Shell	58	186	127	373	157
Montículo	116	264	110	246	104
Rápido	31	55	60	137	37
Mezcla *	99	196	102	195	99
					187

* Véase Apartado 2.31.

Tabla 2.10. Tiempos de ejecución de programas de ordenación (claves con datos asociados).

Hay que hacer notar que:

- El método de selección directa ha mejorado significativamente y resulta el mejor método directo.
- El método de la burbuja continúa siendo el peor método por un ancho margen y sólo su versión mejorada, el método de la sacudida, es ligeramente peor en el caso de array en orden inverso.
- El método rápido ha fortalecido su posición como el más veloz y aparece como el mejor método de ordenación de arrays con diferencia.

2.3. ORDENACION DE FICHEROS SECUENCIALES

2.3.1. Método de mezcla directa

Desgraciadamente, los algoritmos de ordenación presentados en el apartado precedente son inaplicables si la masa de datos a ordenar no cabe en la memoria principal del computador y se encuentra representada, por ejemplo, en un dispositivo periférico de almacenamiento secuencial tal como es una cinta. En este caso se describen los datos en forma de fichero (secuencial) cuya característica es que en cada momento uno, y sólo uno de las componentes, es accesible directamente. Esta es una restricción importante, comparada con las posibilidades que ofrece la estructura array, y, por ello, hay que utilizar técnicas de ordenación distintas. La más importante es la técnica de *fusión* o *mezcla* («merge»). Por mezcla se entenderá, en proceso de datos, la combinación de dos (o más) secuencias ordenadas en una única secuencia ordenada, obtenida por selección repetida

entre los componentes accesibles en cada momento. La mezcla es una operación mucho más simple que la ordenación y se utiliza como operación auxiliar en procesos más complejos de ordenación secuencial. Un método de ordenación basado en este tipo de proceso, denominado de *mezcla directa*, es el siguiente:

1. Se divide la secuencia *a* en dos mitades llamadas *b* y *c*.
2. Se mezclan *b* y *c* combinando ítems aislados para formar pares ordenados.
3. La secuencia resultante de la mezcla se denomina *a*, y se repiten los pasos 1 y 2, esta vez mezclando los pares ordenados para formar cuádruplos ordenados.
4. Se repiten los pasos anteriores, fundiendo los cuádruplos ordenados para formar octuplos ordenados, y se continúa haciendo esto, duplicando cada vez el tamaño de las subsecuencias mezcladas, hasta que la secuencia total quede ordenada.

Por ejemplo, considérese la secuencia

44 55 12 42 94 18 06 67

En el paso 1 el procedimiento de subdivisión produce las secuencias

44 55 12 42

94 18 06 67

La mezcla de componentes aislados (que son secuencias ordenadas de longitud 1) para formar pares ordenados produce

44 94' 18 55' 06 12' 42 67

Subdividiendo nuevamente en partes iguales, y mezclando los pares ordenados se obtiene

06 12 44 94' 18 42 55 67

Una tercera operación de subdivisión y mezcla produce, finalmente, el resultado deseado

06 12 18 42 44 55 67 94

Cada operación que trata una vez el conjunto de datos en su totalidad se denomina una *fase*, y el subproceso más pequeño que, por repetición, constituye el proceso de ordenación se denomina *pasada*.

En el ejemplo anterior, la ordenación necesitó tres pasadas, cada pasada constituida por una fase de subdivisión y otra de mezcla. Para llevar a cabo el

proceso de ordenación son necesarias tres cintas; el proceso, por ello, se denomina, *mezcla o fusión con tres cintas*.

Realmente, las fases de subdivisión no contribuyen a la ordenación, ya que no permutan los ítems de ninguna forma; en cierto sentido, son improductivas, aunque constituyen la mitad de todas las operaciones de copiado. Pueden eliminarse totalmente combinando las fases de subdivisión y mezcla. En vez de mezclar produciendo una secuencia única, el resultado del proceso de mezcla se redistribuye inmediatamente sobre dos cintas, que constituyen la base de la pasada subsiguiente. En contraste con el anterior proceso de dos fases, este método se denomina *mezcla de fase única* o *mezcla compensada*. Evidentemente es superior porque son necesarias solamente la mitad de las operaciones de copia; el precio de esta mejoría es una cuarta cinta.

A continuación se desarrollará un programa de mezcla detalladamente. Los datos se representan, inicialmente, por un array que, sin embargo, se inspecciona en forma *estrictamente secuencial*. Una versión posterior del método de mezcla se basará en la estructura de fichero, lo que permitirá comparar los dos programas y poner de relieve cómo la forma de un programa depende fuertemente de la manera de representar los datos que procesa.

Puede usarse fácilmente un único array en lugar de dos ficheros, si se le considera como una secuencia con dos extremos. En lugar de realizar la mezcla a partir de dos ficheros, pueden tomarse artículos de ambos extremos del array. De esta manera, la forma general de la fase combinada de mezcla y subdivisión puede ilustrarse por la Fig. 2.12. El destino de los ítems mezclados se modifica después de cada par ordenado en la primera pasada, cada cuádruplo ordenado en la segunda pasada, etc., llenando así alternativamente las dos secuencias de destino, representadas por ambos extremos de un array único. Después de cada pasada, ambos arrays intercambian sus papeles; el de origen se convierte en el de destino, y viceversa.

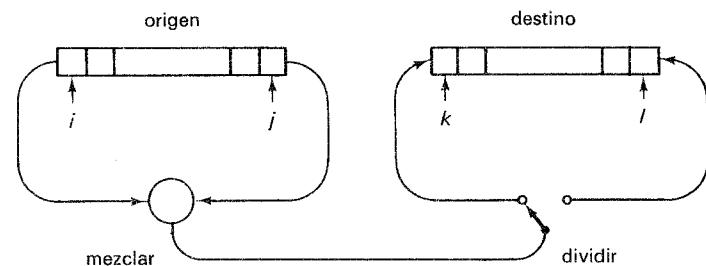


Fig. 2.12. Método de mezcla directa con dos arrays.

Puede conseguirse una simplificación mayor uniendo en un solo array de tamaño doble los dos arrays simples conceptualmente distintos. Así, los datos se representarán por

*a: array[1 .. 2 * n] of item* (2.20)

los índices i, j designan los dos ítems origen mientras k y l denominan los dos destinos (ver Fig. 2.12). Los datos iniciales son, desde luego, los ítems $a_1 \dots a_n$. Evidentemente, se necesita una variable booleana, *arriba*, para designar la dirección del flujo de datos; *arriba = true* significará que, en la pasada en curso, los componentes $a_1 \dots a_n$ se moverán hacia «*arriba*», hacia las variables $a_{n+1} \dots a_{2n}$ mientras *arriba = false* indicará que $a_{n+1} \dots a_{2n}$ deben transferirse hacia «*abajo*», es decir, a_1, \dots, a_n . El valor de *arriba* cambia alternativamente entre pasadas consecutivas. Y, finalmente, se introduce una variable *p* para designar el tamaño de las «subsecuencias» a mezclar. Su valor inicialmente es 1 y se duplica antes de cada pasada sucesiva. Para simplificar, en cierta forma, se supondrá que *n* es una potencia de 2. Así, la primera versión del programa de mezcla directa toma la forma siguiente:

```

procedure mezcladirecta;
  var i, j, k, l: indice;
      arriba: boolean; p: integer;
begin arriba := true; p := 1;
repeat {inicializar indices}
  if arriba then
    begin i := 1; j := n; k := n + 1; l := 2 * n
    end else
    begin k := 1; l := n; i := n + 1; j := 2 * n
    end;
  «mezclar p-uplos de las secuencias i, j en las secuencias k, l»;
  arriba :=  $\neg$  arriba; p := 2 * p
until p = n
end
(2.21)

```

En el paso siguiente se «refina» la instrucción expresada (entre comillas) en lenguaje natural. Evidentemente, esta pasada de mezcla que afecta a *n* ítems es, en sí misma, una sucesión de mezclas de subsecuencias, es decir, de *p*-uplos. Entre cada dos mezclas parciales se cambia el destino pasando del extremo inferior del array al superior o viceversa, con el fin de garantizar una distribución equilibrada entre ambos destinos. Si el destino de los ítems mezclados es el extremo inferior del array, el índice de destino es *k*, que se incrementa en 1 después de cada movimiento de un artículo. Si los ítems han de trasladarse al extremo superior del array de destino, el índice es *l*, que debe disminuirse en 1 después de cada movimiento. Para simplificar la instrucción de mezcla se va a designar el índice de destino como *k* todas las veces, intercambiando los valores de las variables *k* y *l*, y se denominará el incremento a utilizar, todas las veces, por *h*, tomando *h* valores 1 ó -1. Estas consideraciones de diseño conducen al siguiente «refinamiento»:

```

h := 1; m := n; {m = no. de items a mezclar}
repeat q := p; r := p; m := m - 2 * p;
  «mezclar q artículos de i con r artículos de j; el indice
   de destino es k con incremento h»;
  h := -h;
  «intercambiar k y l»;
until m = 0
(2.22)

```

En el nuevo paso de refinamiento hay que formular la instrucción real de mezcla. Aquí hay que tener en cuenta que el final de una subsecuencia que no se ha vaciado en el proceso de mezcla debe añadirse a la sucesión de salida mediante simples operaciones de copia.

```

while (q ≠ 0) ∧ (r ≠ 0) do
begin {seleccionar un item de i o j}
  if a[i] . clave < a[j] . clave then
    begin «mover un item de i a k; avanzar k e i»; q := q - 1
  end else
    begin «mover un item de j a k; avanzar j y k»; r := r - 1
  end
  end
  «copiar final de la secuencia i»;
  «copiar final de la secuencia j»
(2.23)

```

Después de este nuevo refinamiento, con la copia del final de las secuencias, quedan estudiados todos los detalles del programa. Antes de escribirlo de forma completa, interesa eliminar la restricción de que *n* sea una potencia de 2. ¿Qué partes del algoritmo resultan afectadas al eliminar esta restricción? Es fácil convencirse de que la mejor manera de tratar este caso más general es seguir el método anterior mientras sea posible. En el ejemplo, esto significa que hay que continuar mezclando *p*-uplos hasta que los restos de las sucesiones origen sean menores que *p*. La única parte afectada por la eliminación de la restricción está constituida por las instrucciones que determinan los valores de *q* y *r*, longitudes de las secuencias a mezclar. Las siguientes cuatro instrucciones sustituyen las tres

*q := p; r := p; m := m - 2 * p*

y, el lector debe convencerse de ello, representan una implementación efectiva de la estrategia especificada anteriormente; obsérvese que *m* designa el número total de ítems en las dos sucesiones origen que quedan por mezclar:

```

if m > p then q := p else q := m; m := m - q;
if m > p then r := p else r := m; m := m - r;

```

Además, para garantizar la terminación del programa, debe modificarse la condición $p = n$, que controla la repetición «externa», y cambiarla a $p \geq n$. Después de estas modificaciones puede describirse completamente el algoritmo en forma de programa (ver Programa 2.13).

Programa 2.13. Método de mezcla directa.

```

procedure mezcladirecta;
  var i, j, k, l, t: indice;
  h, m, p, q, r: integer; arriba: boolean;
  {observese que a tiene indices 1 . . . 2 * n}
begin arriba := true; p := 1;
repeat h := 1; m := n;
  if arriba then
    begin i := 1; j := n; k := n + 1; l := 2 * n
    end else
    begin k := 1; l := n; i := n + 1; j := 2 * n
    end;
  repeat {mezclar un tramo de i, j en k}
    {q = longitud del tramo i; r = longitud del tramo j}
    if m ≥ p then q := p else q := m; m := m - q;
    if m ≥ p then r := p else r := m; m := m - r;
    while (q ≠ 0) ∧ (r ≠ 0) do
      begin {mezclar}
        if a[i] .clave < a[j] .clave then
          begin a[k] := a[i]; k := k + h; i := i + 1; q := q - 1
          end else
          begin a[k] := a[j]; k := k + h; j := j - 1; r := r - 1
          end
        end;
      {copiar el final del tramo j}
      while r ≠ 0 do
        begin a[k] := a[j]; k := k + h; j := j - 1; r := r - 1
        end;
      {copiar el final del tramo i}
      while q ≠ 0 do
        begin a[k] := a[i]; k := k + h; i := i + 1; q := q - 1
        end;
      h := -h; t := k; k := l; l := t
    until m = 0;
    arriba := ¬arriba; p := 2 * p
  until p ≥ n;
  if ¬arriba then
    for i := 1 to n do a[i] := a[i + n]
end {mezcladirecta}

```

Análisis de la mezcla directa. Como cada pasada duplica p y, dado que el proceso se termina tan pronto como $p \geq n$, el número de pasadas será $\lceil \log_2 n \rceil$. Cada pasada, por definición, copia el conjunto total de n ítems una vez. Por tanto, el número total de movimientos es, exactamente

$$M = n \cdot \lceil \log n \rceil \quad (2.24)$$

El número C de comparaciones entre claves es aún menor que M , ya que no se realizan comparaciones en las operaciones de copia de las partes finales. Sin embargo, dado que la técnica de mezcla se utiliza habitualmente con dispositivos de almacenamiento periférico, el esfuerzo computacional necesario para las operaciones de movimiento o transferencia supera en varios órdenes de magnitud el esfuerzo requerido por las operaciones de comparación. Por ello, el análisis detallado del número de comparaciones es poco interesante desde un punto de vista práctico.

El algoritmo de ordenación por mezcla, aparentemente, es comparable a las técnicas avanzadas estudiadas en el apartado anterior. Sin embargo, la sobre-carga (*overhead*) producida por la manipulación de índices es relativamente alta, y el inconveniente decisivo es la necesidad de almacenar $2n$ ítems. Por esta razón, el procedimiento de mezcla se utiliza raramente con arrays, es decir, con datos almacenados en memoria principal. En las últimas líneas de las Tablas 2.9 y 2.10 aparecen elementos de comparación del comportamiento en tiempo real del método de mezcla directa. La comparación resulta favorable en relación con el método del montículo, pero desfavorable en relación con el método rápido.

2.3.2. Mezcla natural

En el método de mezcla directa, no se obtiene ninguna ventaja cuando los datos están parcialmente ordenados al empezar el proceso. La longitud de las subsecuencias mezcladas en la pasada k -ésima es (menor o) igual a 2^k , no teniendo en cuenta que puede haber subsecuencias más largas ya ordenadas que también podrían mezclarse. De hecho, dos subsecuencias ordenadas cualesquiera de longitudes m y n pueden mezclarse para producir una secuencia única de $m + n$ ítems. El método de mezcla que, en todo momento, mezcla las dos subsecuencias más largas posibles se denomina *método de mezcla natural*.

Una subsecuencia ordenada se denomina a menudo una *tira*. Sin embargo, dado que esta palabra se utiliza más frecuentemente para describir secuencias de caracteres se utilizará la palabra *tramo* en lugar de *tira* para las subsecuencias ordenadas. Se denomina *tramo máximo*, o, para abreviar tramo, a una subsecuencia $a_i \dots a_j$ tal que

$$\begin{aligned} a_k &\leq a_{k+1} && \text{para } k = i \dots j-1 \\ a_{j-1} &> a_j \\ a_j &= a_{j+1} \end{aligned} \quad (2.25)$$

Un método natural de mezcla, por tanto, funde tramos (máximos) en lugar de secuencias de tamaño fijo predeterminado. Los tramos tienen la propiedad de que si dos secuencias de n tramos se mezclan, se produce una secuencia de n tramos exactamente. Por ello, el número total de tramos se divide por dos en cada pasada, y el número de movimientos necesarios de ítems es, en el caso más desfavorable, $n \cdot \lceil \log_2 n \rceil$, pero en el caso medio es aún menor. El número de comparaciones esperadas, sin embargo, es mucho mayor, porque, además de las comparaciones necesarias para la selección de ítems, se necesitan comparaciones adicionales entre ítems consecutivos de cada fichero para determinar el final de cada tramo.

El siguiente ejercicio de programación desarrolla un algoritmo de mezcla natural en la misma forma gradual (paso a paso) que se empleó en el algoritmo de mezcla directa. Se utiliza la estructura de fichero secuencial en lugar de la de array y el algoritmo constituye un método de mezcla con tres cintas, en dos fases no equilibrado. Se supone que la secuencia inicial de ítems dada es el fichero c , en el que aparecerá el resultado ordenado (naturalmente, en las aplicaciones reales de proceso de datos, los datos iniciales se copian primeramente de la cinta original al fichero c , por razones de seguridad). Las dos cintas auxiliares son a y b . Cada pasada consiste en una fase de distribución que reparte equitativamente los tramos de c sobre a y b , y una fase que mezcla los tramos de a y b sobre c . Este proceso se ilustra en la Fig. 2.13.

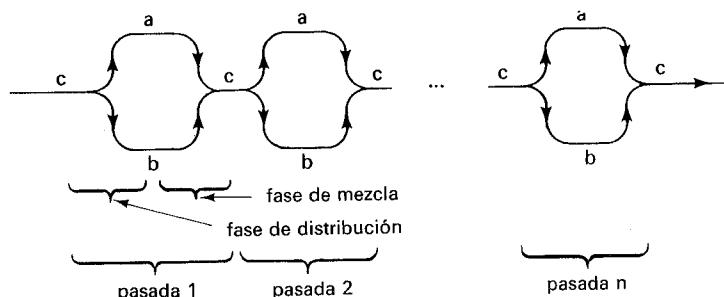


Fig. 2.13. Fases y pasadas del proceso de ordenación.

17	31'	5	59'	13	41	43	67'	11	23	29	47'	3	7	71'	2	19	57'	37	61
5	17	31	59'	11	13	23	29	41	43	47	67'	2	3	7	19	57	71'	37	61
5	11	13	17	23	29	31	41	43	47	59	67'	2	3	7	19	37	57	61	71
2	3	5	7	11	13	17	19	23	29	31	37	41	43	47	57	59	61	67	71

Tabla 2.11. Ejemplo de ordenación por mezcla natural.

Como ejemplo, la Tabla 2.11 muestra el fichero c en su estado original (línea 1) y después de cada pasada (líneas 2-4) de un proceso de mezcla natural aplicado a 20 números. Obsérvese que solo se necesitan tres pasadas. El proceso termina tan pronto como el número de tramos de c es 1 (se supone que existe al menos

un tramo no vacío en el fichero inicial). A tal fin, se usa una variable l que cuenta el número de tramos mezclados sobre el fichero c . Si se definen las entidades globales.

```
type cinta = file of item;
var c: cinta;
```

(2.26)

el programa puede formularse de la manera siguiente:

```
procedure namezcla;
  var l: integer;
      a, b: cinta;
begin
  repeat rewrite(a); rewrite(b); reset(c);
    distribucion;
    reset(a); reset(b); rewrite(c);
    l := 0; mezcla
  until l = 1
end
```

(2.27)

Cada fase aparece claramente como una instrucción diferente. A continuación se refinan ambas, es decir, se expresan más detalladamente. Estas descripciones más detalladas pueden sustituir las instrucciones abreviadas, o bien pueden ser descritas como procedimientos independientes; de esta forma, las instrucciones abreviadas de (2.27) pueden considerarse como llamadas a los procedimientos. Se elige este último método y se define:

```
procedure distribucion; {desde c hacia a y b}
begin
  repeat copitramo(c, a);
    if ¬eof(c) then copitramo(c, b)
  until eof(c)
end
```

(2.28)

```
procedure mezcla;
begin {desde a y b hacia c}
  repeat meztramo; l := l + 1
  until eof(b);
  if ¬eof(a) then
    begin copitramo(a, c); l := l + 1
    end
end
```

(2.29)

Este método de distribución se supone que produce, bien igual número de tramos en los ficheros *a* y *b*, o bien un tramo más en el fichero *a* que en el *b*. Dado que los pares de tramos correspondientes se mezclan entre sí, puede quedar un último tramo en el fichero *a*, que haya que copiar simplemente. Los procedimientos *mezcla* y *distribucion* están formulados en base a los procedimientos subordinados *meztramo* y *copitramo* cuyas tareas son obvias. A continuación se expresan estos procesos más detalladamente; necesitan la introducción de una variable booleana global *fdt* que especifica si se ha llegado o no al final del tramo.

```
procedure copitramo(var x, y: cinta);
begin {copiar un tramo de x a y}
  repeat copiar(x, y) until fdt
end
```

```
procedure meztramo;
begin {mezclar un tramo de a y b sobre c}
  repeat if a↑ .clave < b↑ .clave then
    begin copiar(a, c);
      if fdt then copitramo(b, c)
    end else
    begin copiar(b, c);
      if fdt then copitramo(a, c)
    end
  until fdt
end
```

(2.31)

El proceso de comparación y selección de claves al mezclar un tramo termina tan pronto como se agota alguno de los tramos. Después, el otro tramo (todavía no agotado) debe transferirse al tramo resultante por simple copia de su parte final. Esto se hace llamando al procedimiento *copitramo*.

Los dos procedimientos se apoyan en el procedimiento subordinado *copiar*, que transfiere un ítem de un fichero origen *x* a un fichero destino *y* determinando, además, si se ha llegado al final de un tramo. Para determinar el final de un tramo hay que retener la clave del último ítem leído (copiado) para compararla con la del ítem siguiente. Este «mirar adelante» se lleva a efecto por inspección de la variable buffer del fichero, *x*↑.

```
procedure copiar(var x, y: cinta);
  var buf: item;
begin read(x, buf); write(y, buf);
  if cof(x) then fdt := true else fdt := buf .clave > x↑ .clave
end
```

(2.32)

Con esto se remata el proceso gradual de construcción del procedimiento de

ordenación por mezcla natural. Lamentablemente, el programa es incorrecto, como habrá observado el lector atento, ya que en algunos casos no ordena en forma adecuada. Considérese, por ejemplo la siguiente sucesión de datos de entrada

3 2 5 11 7 13 19 17 23 31 29 37 43 41 47 59 57 61 71 67

al distribuir tramos consecutivos sobre los ficheros *a* y *b*, alternativamente, se obtiene:

a = 3' 7 13 19' 29 37 43' 57 61 71'

b = 2 5 11' 17 23 31' 41 47 59' 67

Estas secuencias se mezclan en forma inmediata en un único tramo con lo que el proceso se daría por terminado. El ejemplo, aunque no conduce a un comportamiento erróneo del programa pone de relieve que la simple distribución de los tramos en varios ficheros puede producir un número de tramos resultantes menor que el número de tramos originales. Esto se debe a que el primer artículo del tramo *i* + 2-ésimo puede ser mayor que el último ítem del tramo *i*-ésimo y, por ello, dar lugar a que ambos tramos se mezclen automáticamente en un único tramo.

Aunque se supone que el procedimiento *distribucion* reparte equitativamente los tramos que produce entre los dos ficheros, una consecuencia importante es que el número real de tramos resultantes en *a* y *b* puede diferir significativamente. El procedimiento descrito de mezcla, sin embargo, une solamente pares de tramos y termina cuando se llega al final del fichero *b*, perdiéndose por ello la parte extrema de uno de los ficheros. Considérense los siguientes datos de entrada que se ordenan (y truncan) en dos pasadas:

17	19	13	57	23	29	11	59	31	37	7	61	41	43	5	67	47	71	2	3
13	17	19	23	29	31	37	41	43	47	57	71	11	59						
11	13	17	19	23	29	31	37	41	43	47	57	59	71						

Tabla 2.12. Resultado incorrecto del programa de mezcla.

Este tipo de error de programación se presenta en muchas ocasiones. El error se debe a la inadvertencia de una de las posibles consecuencias de una operación presumiblemente simple. Como casi siempre que se presenta, hay varias formas de remediar el error y se debe elegir una de ellas. A menudo se presentan dos posibilidades que difieren de manera fundamental:

1. Se admite que la operación de distribución está programada incorrectamente y no cumple la condición de que el número de tramos sea igual (o difiera como

mucho en 1) en ambos ficheros. Se mantiene el esquema original de operación y se corrige adecuadamente el procedimiento erróneo.

2. Se comprende que la corrección de la parte errónea produce modificaciones de largo alcance y se intenta encontrar formas de cambiar otras partes del algoritmo para adaptar la parte incorrecta sin modificarla.

En general, el primer camino parece ser el más seguro y elegante, es la forma más honesta, y aporta un cierto grado de inmunidad contra las consecuencias ulteriores de efectos secundarios inadvertidos. Por ello es el camino generalmente recomendado (con razón) para encontrar una solución.

Sin embargo, hay que hacer notar que no debe ignorarse totalmente la segunda posibilidad. Por esta razón, se elabora y se ilustra más este ejemplo modificando el proceso de mezcla en lugar de modificar el proceso de distribución que es realmente el erróneo.

Es decir, se deja intacto el esquema de distribución y se renuncia a la condición de que los tramos estén distribuidos equitativamente. Esto puede producir un rendimiento menos óptimo. Sin embargo, el funcionamiento en el caso más desfavorable no se modifica, y, además, el caso de distribución altamente desequilibrada es, estadísticamente, *poco probable*. Por tanto, las consideraciones de eficiencia no son argumentos serios contra esta solución.

Si la condición de distribución equilibrada de tramos no se cumple, el proceso de mezcla debe modificarse para que una vez alcanzado el final de un fichero, se copie *todo* el final del otro en vez de un tramo como máximo.

Esta modificación es inmediata, y muy simple, en comparación con cualquier cambio en el esquema de distribución (se insta al lector a que se convenza de la verdad de esta afirmación). La versión revisada del algoritmo de mezcla se incluye en el Programa completo 2.14

Programa 2.14. Método de mezcla natural.

```
program mezclanatural (input, output);
{ordenación por mezcla natural, 3-cintas, 2-fases}
type item = record clave: integer
           {aqui pueden definirse otros campos}
           end;
cinta = file of item;
var c: cinta; n: integer; buf: item;
procedure listar (var f: cinta);
  var x: item;
begin reset(f);
  while not eof(f) do
    begin read(f, x); write(output, x .clave)
    end;
  writeln
end;
```

```
end {listar};
procedure namezcla;
  var l: integer; {no. de tramos a mezclar}
  fdt: boolean; {indicador de fin de tramo}
  a, b: cinta;
procedure copiar(var x, y: cinta);
  var buf: item;
begin read(x, buf); write(y, buf);
  if eof(x) then fdt := true else fdt := buf .clave > x↑ .clave
end;
procedure copitramo (var x, y: cinta);
begin {copiar un tramo de x a y}
  repeat copiar(x, y) until fdt
end;
procedure distribución;
begin {desde c hasta a y b}
  repeat copitramo (c, a);
    if not eof(c) then copitramo(c, b)
    until eof(c)
end;
procedure meztramo;
begin {desde a y b a c}
  repeat
    if a↑ .clave ≤ b↑ .clave then
      begin copiar (a, c);
        if fdt then copitramo (b, c)
      end
    else
      begin copiar (b, c);
        if fdt then copitramo (a, c)
      end
    until fdt
  end;
procedure mezcla;
begin {desde a y b a c}
  while not eof(a) and not eof(b) do
    begin meztramo; l := l + 1
    end;
  while not eof(a) do
    begin copitramo (a, c); l := l + 1
    end;
  while not eof(b) do
    begin copitramo (b, c); l := l + 1
    end;
end;
```

Programa 2.14. (Continuación)

```

    end;
    listar(c)
end;
begin
repeat rewrite(a); rewrite(b); reset(c);
  distribucion;
  reset(a); reset(b); rewrite(c);
  l := 0; mezcla;
until l = 1
end;
begin {programa principal; leer la secuencia de entrada que termina con un 0}
  rewrite(c); read(buf .clave);
repeat write(c, buf); read(buf .clave)
until buf .clave = 0;
listar(c);
namezcla;
listar(c)
end.

```

Programa 2.14. (Continuación)

2.3.3. Mezcla equilibrada múltiple

El esfuerzo necesario para realizar ordenación secuencial es proporcional al número de pasadas, dado que, por definición, cada pasada necesita copiar el conjunto total de datos. Una forma de reducir el número de pasadas es distribuir los tramos sobre más de dos ficheros. La mezcla de t tramos equitativamente distribuidos en N cintas produce una sucesión de t/N tramos. Una segunda pasada reduce su número a t/N^2 , una tercera a t/N^3 y, tras k pasadas, quedan t/N^k tramos. El número total de pasadas necesarias para ordenar n artículos mediante *mezcla N-uple* es, por ello, $k = \lceil \log_N n \rceil$. Como cada pasada requiere n operaciones de copia, el número total de éstas, en el caso más desfavorable, es

$$M = n \cdot \lceil \log_N n \rceil$$

El siguiente ejercicio de programación a desarrollar es un proceso de ordenación por mezcla múltiple. Para poner más de relieve las diferencias de este programa con el anterior de mezcla en dos fases, se formulará el proceso de mezcla múltiple en una sola fase. Esto obliga a que en cada pasada haya un número igual de ficheros de entrada y salida sobre los que se distribuyan alternativamente los tramos consecutivos. Utilizando N ficheros, el algoritmo será por lo tanto una mezcla $N/2$ -uple, supuesto N par. Siguiendo la estrategia previamente adoptada, no hace falta detectar la mezcla automática de dos tramos consecutivos distribuidos sobre

la misma cinta. Por tanto, es preciso diseñar el programa de mezcla sin suponer números de tramos estrictamente iguales en las cintas de entrada.

En este programa aparece por primera vez una aplicación justificada de la estructura de datos formada por un array de ficheros. Realmente sorprenden las importantes diferencias entre este programa y el anterior debidas al paso de mezcla doble a mezcla múltiple. Las diferencias son debidas principalmente a que el proceso de mezcla ya no puede terminarse simplemente cuando uno de los tramos de entrada llega a su final. En lugar de ello debe mantenerse una lista de entradas todavía activas, es decir, aún no agotadas. Otra complicación deriva del hecho de necesitar que se haga el intercambio de función de los grupos de cintas de entrada y salida después de cada pasada.

Se empieza por definir, además de los tipos familiares *ítem* y *cinta*, el tipo

$$\text{nocinta} = 1 \dots N \quad (2.33)$$

Obviamente, los números de cinta se utilizan como índices del array de ficheros de ítems. Supóngase que la secuencia inicial de ítems viene dada por la variable

$$f0: \text{cinta} \quad (2.34)$$

y que hay N cintas disponibles para el proceso de ordenación, siendo N par.

$$f: \text{array } [\text{nocinta}] \text{ of cinta} \quad (2.35)$$

Una técnica recomendable para tratar el problema de cambio de función de las cintas (de entrada a salida) es introducir una función de correspondencia entre índices de cinta. En lugar de acceder directamente a una cinta por su índice i , se accede mediante una tabla c , es decir, en lugar de

$$f[i] \text{ se escribe } f[c[i]]$$

la función de correspondencia se define por tanto de la forma:

$$c: \text{array } [\text{nocinta}] \text{ of nocinta} \quad (2.36)$$

Si inicialmente $c[i] = i$ para todo i un cambio de función en las cintas puede describirse simplemente cambiando entre sí los componentes de la tabla

$$c[1] \leftrightarrow c[n2 + 1]$$

$$c[2] \leftrightarrow c[n2 + 2]$$

...

$$c[n2] \leftrightarrow c[n]$$

siendo $n_2 = n/2$. Por tanto pueden considerarse siempre

$$f[c[1]], \dots, f[c[n_2]]$$

como cintas de entrada y

$$f[c[n_2 + 1]], \dots, f[c[n]]$$

como cintas de salida (en lo que sigue $f[c[j]]$ se llamará simplemente «cinta j » dentro de los comentarios). El algoritmo puede ahora formularse inicialmente de la forma siguiente:

```

procedure mezclacinta;
  var i, j: nocinta;
    l: integer; {no. de tramos distribuidos}
    c: array [nocinta] of nocinta;
begin {distribuir los tramos iniciales en c[1] ... c[n_2]}
  j := n_2; l := 0;
  repeat if j < n_2 then j := j + 1 else j := 1;
    «copiar un tramo de f0 a cinta j»;
    l := l + 1
  until eof(f0);
  for i := 1 to n do c[i] := i;
  repeat {mezcla de c[1] ... c[n_2] a c[n_2 + 1] ... c[n]}
    «inicializar cintas de entrada»;
    l := 0;
    j := n_2 + 1; {j = indice de cinta de salida}
    repeat l := l + 1;
      «mezclar un tramo desde las entradas a c[j]»
      if j < n then j := j + 1 else j := n_2 + 1
    until «todas las entradas agotadas»;
    «cambiar la función de las cintas»
  until l = 1;
  {la cinta ordenada es c[1]}
end

```

(2.37)

En primer lugar se refina la operación de copia utilizada en la distribución inicial de tramos; para ello se introduce nuevamente una variable auxiliar como buffer del último ítem leído:

buf: item

y se sustituye «copiar un tramo de $f0$ a cinta j » por la instrucción:

```

repeat read(f0, buf);
  write(f[j], buf)
until (buf .clave > f0↑ .clave) ∨ eof(f0)

```

(2.38)

La copia de un tramo termina cuando, bien se encuentra el primer artículo del tramo siguiente ($buf .clave > f0↑ .clave$), o bien se alcanza el final del fichero de entrada ($eof(f0)$).

Quedan por especificar más detalladamente en el algoritmo las instrucciones:

1. Inicializar cintas de entrada.
2. Mezclar un tramo desde las entradas a $c[j]$.
3. Cambiar la función de las cintas

y el predicado

4. Todas las entradas agotadas

En primer lugar deben identificarse con más precisión los ficheros de entrada en curso de proceso. En particular, el número de ficheros de entrada «activos» puede ser menor que $n/2$. En efecto, puede haber como máximo tantos ficheros origen como tramos; el proceso de ordenación termina tan pronto queda un único fichero. Esto deja abierta la posibilidad de que al comenzar la última pasada de ordenación haya menos de n_2 tramos. Por ello se introduce una variable, k_1 , para designar el número real de ficheros de entrada utilizados. Se incorpora la inicialización de k_1 en la instrucción «inicializar las cintas de entrada» de la forma siguiente:

```

if l < n_2 then k1 := l else k1 := n_2
for i := 1 to k1 do reset(f[c[i]]);

```

Naturalmente, la instrucción 2 deberá disminuir k_1 cuando se termine un fichero de entrada. De aquí que el predicado 4 pueda expresarse fácilmente mediante la relación

$$k1 = 0$$

La instrucción 2 es más difícil de refinar; consiste en la selección repetida de la mínima clave entre las fuentes (ficheros de entrada) disponibles y el transporte subsiguiente a su destino, es decir, la cinta de salida en curso en el proceso. El proceso se complica, además, por la necesidad de determinar el final de cada tramo. Este final puede alcanzarse porque (a) la clave siguiente es inferior a la clave en curso, o (b) se alcanza el final del fichero de entrada. En este último caso se elimina la cinta disminuyendo k_1 ; en el primer caso el tramo se cierra excluyendo al fichero de ulteriores selecciones de ítems, pero solamente hasta que se complete el tramo de salida que se está procesando. Esto hace obvia la necesidad de una

segunda variable, k_2 , para designar el número de cintas origen que están disponibles en cada momento para seleccionar el ítem siguiente. Este valor se hace inicialmente igual a k_1 y se disminuye siempre que se termina un tramo como consecuencia de la condición (a).

Desgraciadamente, la introducción de k_2 no es suficiente; el conocimiento del número de cintas no basta. Es preciso conocer exactamente qué cintas quedan en juego. Una solución obvia es emplear un array con componentes booleanos que indiquen la disponibilidad de las cintas. Sin embargo, se adopta un método diferente que permite un procedimiento de selección más eficiente, ya que, al fin y al cabo, es la parte que más se repite durante todo el algoritmo. En vez de utilizar un array booleano, se introduce una segunda tabla de correspondencia de cintas representada por la variable cd . Esta tabla se emplea en lugar de c de forma que $cd[1] \dots cd[k_2]$ son los índices de las cintas disponibles. De acuerdo con esto, la instrucción 2 puede formularse como sigue:

```

k2 := k1;
repeat «seleccionar la clave mínima, sea  $cd[mx]$  su numero de cinta»;
  read(f[cd[mx]], buf);
  write(f[c[j]], buf);
  if eof(f[cd[mx]]) then «eliminar cinta» else
    if buf .clave > f[cd[mx]]↑ .clave then «cerrar tramo»
  until k2 = 0

```

(2.39)

Dado que el número de cintas disponible en cualquier instalación de proceso de datos es normalmente bastante reducido, el algoritmo de selección, especificado más detalladamente en el siguiente paso de refinamiento, puede bien ser una búsqueda lineal directa. La instrucción «eliminar cinta» implica una disminución de k_1 lo mismo que de k_2 , y la re-asignación de índices en la tabla cd . La instrucción «cerrar tramo» se limita a disminuir k_2 y reorganizar los componentes de cd de forma adecuada. Los detalles se muestran en el Programa 2.15 que es el último paso del proceso de refinamiento gradual que va desde (2.37) hasta (2.39). Obsérvese que las cintas se rebobinan mediante el procedimiento *rewrite* en cuanto se ha leído su último tramo. La instrucción «cambio de función de cintas» se elabora de acuerdo con las explicaciones dadas anteriormente.

Programa 2.15. Ordenación por mezcla equilibrada.

```

program mezclaequilibrada (output);
{ordenacion por mezcla equilibrada n-uple}
const n = 6; n2 = 3; {no. de cintas}
type item = record
  clave: integer
end;
cinta file of item;
nocinta 1 .. n;
var long, alea: integer; {usados para generar fichero}

```

```

fdc: boolean; {fin de cinta}
buf: item;
f0: cinta; {f0 es la cinta de entrada con números aleatorios}
f: array [1 .. n] of cinta;
procedure listar(var f: cinta; n: nocinta);
  var z: integer;
begin writeln('CINTA', n: 2); z := 0;
while ¬eof(f) do
  begin read(f, buf); write(output, buf .clave: 5); z := z + 1;
  if z = 25 then
    begin writeln(output); z := 0;
    end
  end;
  if z ≠ 0 then writeln (output); reset(f)
end {listar};

procedure mezclacinta;
  var i, j, mx, cx: nocinta;
  k1, k2, l; integer;
  x, min: integer;
  c, cd: array [nocinta] of nocinta;
begin {distribuir los tramos iniciales en c[1] ... c[n2]}
  for i := 1 to n2 do rewrite(f[i]);
  j := n2; l := 0;
  repeat if j < n2 then j := j + 1 else j := 1;
    {copiar un tramo desde f0 a la cinta j}
    l := l + 1;
    repeat read(f0, buf); write(f[j], buf)
    until (buf .clave > f0↑ .clave) ∨ eof(f0)
  until eof(f0);
  for i := 1 to n do c[i] := i;
  repeat {mezclar desde c[1] ... c[n2] hacia c[n2 + 1] ... c[n]}
    if l < n2 then k1 := l else k1 := n2;
    {k1 = no. de cintas de entrada en esta fase}
    for i := 1 to k1 do
      begin reset(f[c[i]]); listar(f[c[i]], c[i]); cd[i] := c[i]
      end;
    l := 0; {l = numero de tramos mezclados}
    j := n2 + 1; {j = indice de la cinta de salida}
    repeat {mezclar un tramo desde c[1] ... c[k1] hacia c[j]}
      k2 := k1; l := l + 1; {k2 = no. de cintas de entrada activas}
      repeat {seleccionar el elemento mínimo}

```

Programa 2.15. (Continuación)

```

i := 1; mx := 1; min := f[cd[1]]↑ .clave;
while i < k2 do
  begin i := i + 1; x := f[cd[i]]↑ .clave;
    if x < min then
      begin min := x; mx := i
      end
    end;
  {cd[mx] tiene el elemento mínimo, que se transfiere a c[j]}
  read(f[cd[mx]], buf); fdc := eof(f[cd[mx]]);
  write(f[c[j]], buf);
  if fdc then
    begin rewrite(f[cd[mx]]); {eliminar cinta}
      cd[mx] := cd[k2]; cd[k2] := cd[k1];
      k1 := k1 - 1; k2 := k2 - 1
    end else
    if buf .clave > f[cd[mx]]↑ .clave then
      begin cx := cd[mx]; cd[mx] := cd[k2]; cd[k2] := cx;
      k2 := k2 - 1
      end
    until k2 = 0;
    if j < n then j := j + 1 else j := n2 + 1
  until k1 = 0;
  for i := 1 to n2 do
    begin cx := c[i]; c[i] := t[i + n2]; t[i + n2] := cx
    end
  until l = 1;
  reset(f[c[1]]); listar(f[c[1]], c[1]); {el resultado ordenado esta en c[1]}
end {mezclacinta};

begin {generar el f_hero aleatorio f0}
  long := 200; alea := 7789; rewrite(f0);
  repeat alea := (131071 * alea) mod 2147483647;
    buf .clave := alea div 2147484; write(f0, buf); long := long - 1
  until long = 0;
  reset(f0); listar(f0, 1);
  mezclacinta
end.

```

Programa 2.15. (Continuación)

2.3.4. Ordenación polifásica

Hasta aquí se han estudiado las técnicas necesarias, y se cuenta con una base apropiada, como para investigar y programar otros algoritmos de ordenación

cuya eficacia es superior al método de mezcla equilibrada. Se ha visto que la mezcla equilibrada elimina las operaciones, meramente de copia, necesarias cuando las operaciones de distribución y mezcla están juntas en una fase única. La cuestión que se plantea es si las cintas dadas podrían utilizarse aún mejor. Resulta que esto último sí es posible; para conseguir esta nueva mejora hay que abandonar la noción rígida de pasada, es decir, usar las cintas de una forma más sofisticada que la de tener siempre $N/2$ cintas origen e igual número de cintas destino, e intercambiar las cintas origen y destino al final de cada pasada. En vez de esto, se difumina la noción de pasada. El método fue inventado por R. L. Gilstad [2-3] y fue bautizado con el nombre de *método de ordenación polifásica (Polyphase Sort)*.

El método se ilustra primeramente con un ejemplo que emplea tres cintas. En todo momento los ítems se mezclan desde dos cintas sobre una tercera. Cada vez que una de las cintas origen llega a su final se convierte inmediatamente en la cinta destino de las operaciones de mezcla desde la cinta aún no terminada y la cinta que previamente era de destino.

Como se sabe que n tramos en cada cinta de entrada se transforman en n tramos en la cinta de salida, sólo se necesita tener en cuenta el número de tramos presentes en cada cinta (en lugar de especificar las propias claves). En la figura 2.14 se supone que las cintas *f1* y *f2* contienen 13 y 8 tramos, respectivamente. De esta forma, en la primera «pasada» se mezclan ocho tramos desde *f1* y *f2* a *f3*, en la segunda «pasada» los restantes 5 tramos se mezclan desde *f3* y *f1* sobre *f2*, etcétera. Al final *f1* es el fichero ordenado

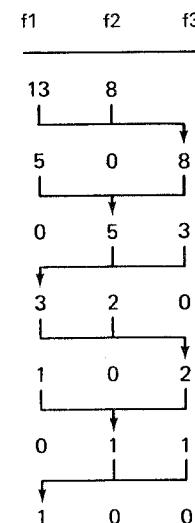


Fig. 2.14. Método polifásico para 21 tramos con tres cintas.

Un segundo ejemplo muestra el método polifásico con 6 cintas. Supóngase que, inicialmente, se tienen *f1* con 16 tramos, *f2* con 15, *f3* con 14, *f4* con 12 y

f_5 con 8; en la primera pasada parcial, se mezclan 8 tramos sobre f_6 ; al final f_2 contiene el conjunto de ítems ordenado (ver Fig. 2.15).

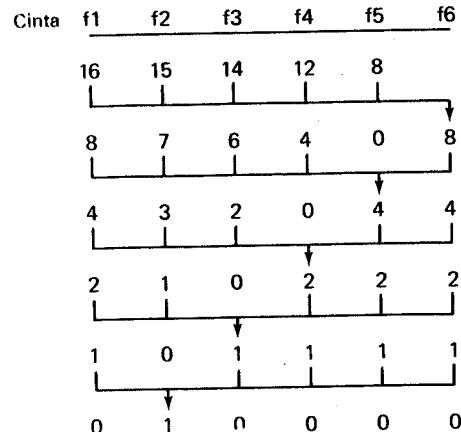


Fig. 2.15. Ordenación por mezcla polifásica de 65 tramos y seis cintas.

El método polifásico es más eficiente que el de mezcla equilibrada —fijadas N cintas—, ya que siempre opera con una mezcla $N - 1$ -uple en lugar de $N/2$ -uple. Como el número de pasadas requerido es, aproximadamente, $\log_N n$, siendo n , el número de ítems a ordenar y N el grado de las operaciones de mezcla, el método polifásico promete una mejora significativa sobre el de mezcla equilibrada. Desde luego, la distribución de los tramos iniciales en los ejemplos anteriores fue elegida cuidadosamente. Para encontrar las distribuciones iniciales de tramos que producen un funcionamiento adecuado, se actúa de atrás adelante, empezando con la distribución final (última línea Fig. 2.15). Si se escriben nuevamente las tablas de los dos ejemplos y se rota cada fila una posición respecto a la fila previa se obtienen las tablas 2.13 y 2.14 para seis pasadas con tres y seis cintas, respectivamente.

i	$a_1^{(i)}$	$a_2^{(i)}$	$\sum a_i^{(i)}$
0	1	0	1
1	1	1	2
2	2	1	3
3	3	2	5
4	5	3	8
5	8	5	13
6	13	8	21

Tabla 2.13. Distribución perfecta de tramos en dos cintas.

i	$a_1^{(i)}$	$a_2^{(i)}$	$a_3^{(i)}$	$a_4^{(i)}$	$a_5^{(i)}$	$\sum a_i^{(i)}$
0	1	0	0	0	0	1
1	1	1	1	1	1	5
2	2	2	2	2	1	9
3	4	4	4	3	2	17
4	8	8	7	6	4	33
5	16	15	14	12	8	65

Tabla 2.14. Distribución perfecta de tramos en cinco cintas.

De la tabla 2.13 pueden deducirse las relaciones

$$\begin{aligned} a_2^{(i+1)} &= a_1^{(i)} \\ a_1^{(i+1)} &= a_1^{(i)} + a_2^{(i)} \end{aligned} \quad \left. \right\} \text{ para } i > 0 \quad (2.40)$$

y $a_1^{(0)} = 1$, $a_2^{(0)} = 0$. Haciendo $a_1^{(0)} = f_{i+1}$, se obtiene

$$\begin{aligned} f_{i+1} &= f_i + f_{i-1}, \quad \text{para } i \geq 1 \\ f_1 &= 1 \\ f_0 &= 0 \end{aligned} \quad (2.41)$$

Estas reglas recursivas (o relaciones de recurrencia) definen los llamados números de Fibonacci:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55$$

Cada número de Fibonacci es la suma de sus dos precedentes. Por tanto, los números de tramos iniciales en ambas cintas deben ser dos números de Fibonacci consecutivos para que el método polifásico funcione con tres cintas adecuadamente. ¿Qué ocurre con el segundo ejemplo (tabla 2.14) de seis cintas? Las reglas de formación se obtienen fácilmente; son

$$\begin{aligned} a_3^{(i+1)} &= a_1^{(i)} \\ a_4^{(i+1)} &= a_1^{(i)} + a_3^{(i)} = a_1^{(i)} + a_1^{(i+1)} \\ a_5^{(i+1)} &= a_1^{(i)} + a_4^{(i)} = a_1^{(i)} + a_1^{(i-1)} + a_1^{(i-2)} + a_1^{(i-3)} \\ a_2^{(i+1)} &= a_1^{(i)} + a_3^{(i)} = a_1^{(i)} + a_1^{(i-1)} + a_1^{(i-2)} + a_1^{(i-3)} \\ a_1^{(i+1)} &= a_1^{(i)} + a_2^{(i)} = a_1^{(i)} + a_1^{(i-1)} + a_1^{(i-2)} + a_1^{(i-3)} + a_1^{(i-4)} \end{aligned} \quad (2.42)$$

poniendo f_i en lugar de $a_1^{(i)}$, se obtiene

$$\begin{aligned} f_{i+1} &= f_i + f_{i-1} + f_{i-2} + f_{i-3} + f_{i-4}, \quad \text{para } i \geq 4 \\ f_4 &= 1 \\ f_i &= 0, \quad \text{para } i < 4 \end{aligned} \quad (2.43)$$

Estos números son los de Fibonacci de orden 4. En general, se definen los *números de Fibonacci de orden p*, de la forma siguiente:

$$\begin{aligned} f_{i+1}^{(p)} &= f_i^{(p)} + f_{i-1}^{(p)} + \cdots + f_{i-p}^{(p)}, \quad \text{para } i \geq p \\ f_p^{(p)} &= 1 \\ f_i^{(p)} &= 0, \quad \text{para } 0 \leq i < p \end{aligned} \tag{2.44}$$

Obsérvese que los números ordinarios de Fibonacci son los de orden 1.

Hasta el momento se ha visto que para aplicar perfectamente el método polifásico con n cintas, los números iniciales de tramos deben ser sumas de sucesiones cualquiera con $n - 1, n - 2, \dots, 1$ (ver tabla 2.15) números consecutivos de

$I \backslash n$	3	4	5	6	7	8
1	2	3	4	5	6	7
2	3	5	7	9	11	13
3	5	9	13	17	21	25
4	8	17	25	33	41	49
5	13	31	49	65	81	97
6	21	57	94	129	161	193
7	34	105	181	253	321	385
8	55	193	349	497	636	769
9	89	355	673	977	1261	1531
10	144	653	1297	1921	2501	3049
11	233	1201	2500	3777	4961	6073
12	377	2209	4819	7425	9841	12097
13	610	4063	9289	14597	19521	24097
14	987	7473	17905	28697	38721	48001
15	1597	13745	34513	56417	76806	95617
16	2584	25281	66526	110913	152351	190465
17	4181	46499	128233	218049	302201	379399
18	6765	85525	247177	428673	599441	755749
19	10946	157305	476449	842749	1189041	1505425
20	17711	289329	918385	1656801	2358561	2998753

Tabla 2.15. Número de tramos que permiten una distribución perfecta.

Fibonacci. Esto implica que, aparentemente, este método sólo es aplicable a datos de entrada cuyo número de tramos sea la suma de $n - 1$ sumandos de Fibonacci. La pregunta importante que se plantea es, ¿qué debe hacerse cuando el número inicial de tramos no coincide con esta suma ideal? La contestación es simple (y típica en casos como éste). Se simula la existencia de tramos vacíos hipotéticos de manera que la suma de los tramos reales e hipotéticos sea el valor ideal. Los tramos vacíos se llaman *tramos ficticios*. Sin embargo, ésta no es una

respuesta satisfactoria, ya que inmediatamente se plantea una pregunta más difícil. ¿Cómo se reconocen los tramos ficticios durante el proceso de mezcla? Antes de contestar esta pregunta hay que investigar primero el problema de la distribución inicial de tramos y decidir la forma de distribuir los tramos reales y ficticios sobre las $n - 1$ cintas.

Para hallar una regla de distribución apropiada, sin embargo, es preciso saber cómo se mezclan los tramos reales y ficticios. Evidentemente, la selección de un tramo ficticio en la cinta i significa exactamente que la cinta i se ignora durante el proceso, produciéndose una mezcla de menos de $n - 1$ cintas origen. Mezclar tramos ficticios procedentes de las $n - 1$ cintas origen equivale a no efectuar realmente ninguna operación de mezcla, sino simplemente a grabar un tramo ficticio en la cinta de salida. De esto se deduce que los tramos ficticios deben distribuirse sobre las $n - 1$ cintas lo más uniformemente posible, ya que interesa realizar mezclas reales del mayor número de cintas origen posible.

Dejando a un lado momentáneamente los tramos ficticios se va a tratar el problema de distribuir un número *cualquier* de tramos sobre $n - 1$ cintas. Está claro que los números de Fibonacci de orden $n - 2$, que especifican el número deseado de tramos por cinta, pueden generarse a medida que avanza el proceso de distribución. Si se supone $n = 6$, por ejemplo, tomando como referencia la tabla 2.14 se empieza distribuyendo los tramos de la forma indicada en la fila de índice $I = 1$ (1, 1, 1, 1, 1); si hay más tramos disponibles, se sigue con la segunda fila (2, 2, 2, 2, 1); si todavía no se han agotado los tramos se continúa distribuyendo según la tercera fila (4, 4, 4, 3, 2) y así sucesivamente. Se denominará *nivel* al índice de *fila*. Evidentemente, cuanto mayor sea el número de tramos, más alto será el nivel de los números de Fibonacci que, casualmente, es igual al número de pasadas de mezcla o cambios de cinta necesarios para la ordenación subsiguiente.

El algoritmo de distribución puede formularse en primera versión de la manera siguiente:

1. Se toman como objetivo de distribución de tramos los números de Fibonacci de orden $n - 2$, nivel 1.
2. Se distribuyen de acuerdo con el objetivo propuesto.
3. Si se alcanza el objetivo, se calcula el siguiente nivel de números de Fibonacci; la diferencia entre éstos y los del nivel anterior constituye el nuevo objetivo de distribución. Se vuelve al paso 2. Si el objetivo no puede alcanzarse porque el número de tramos se acaba, se da por terminado el proceso de distribución.

Las reglas para calcular el nivel siguiente de números de Fibonacci están en la definición (2.44). Por ello puede concentrarse la atención sobre el paso 2 en el que, con un objetivo dado, los tramos siguientes deben distribuirse uno tras otro sobre $n - 1$ cintas. Aquí entran nuevamente en consideración los tramos ficticios.

Supóngase que al pasar de un nivel a otro más alto se describe el siguiente objetivo por las diferencias d_i con $i = 1, \dots, n - 1$, siendo d_i el número de tra-

mos a grabar en la cinta i en el paso en curso. Puede suponerse que, inmediatamente a continuación, se sitúan d_i tramos ficticios sobre la cinta i , y considerar entonces el proceso sucesivo de distribución como una *sustitución* de tramos ficticios por reales, disminuyéndose d_i en 1 cada vez que se produce una sustitución con objeto de anotarla. De esta forma los d_i indican el número de tramos ficticios en la cinta i cuando se acaban los tramos del fichero origen.

No se conoce un algoritmo que produzca la distribución de tramos óptima, pero se ha comprobado que el proceso que se explica a continuación es un método muy bueno. Se denomina de *distribución horizontal* (ver Knuth, vol. 3, pág. 270). Este nombre es explicable si se consideran los tramos apilados en forma de silos tal como indica la Fig. 2.16 para $n = 6$, nivel 5 (ver Tabla 2.14).

Para alcanzar una distribución equilibrada, lo más rápidamente posible, de los tramos ficticios restantes, se reduce el tamaño de las pilas extrayendo tramos ficticios en niveles horizontales de izquierda a derecha. De esta forma los tramos se distribuyen sobre las cintas tal como indica la secuencia de números de la Fig. 2.16.

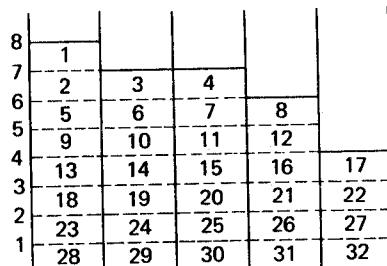


Fig. 2.16. «Distribución horizontal» de tramos.

Sobre esta base es posible describir el algoritmo en forma de un procedimiento denominado *selecinta*, que es llamado cada vez que se ha copiado un tramo y hay que seleccionar una nueva cinta para el tramo siguiente. Se supone una variable j que designa el índice de la cinta de destino en curso de proceso. a_i y d_i designan los números de distribución ideales y ficticios para la cinta i .

```
j: nocinta;
a, d: array [nocinta] of indice;
nivel: integer
```

(2.45)

Estas variables se inicializan con los valores siguientes:

$$\begin{aligned} a_i &= 1, & d_i &= 1 \quad \text{para } i = 1 \dots n - 1 \\ a_n &= 0, & d_n &= 0 \quad (\text{ficticio}) \\ j &= 1 \\ nivel &= 1 \end{aligned}$$

Obsérvese que *selecinta* tiene que calcular la fila siguiente de la Tabla 2.14, es decir, los valores $a_1^{(0)} \dots a_{n-1}^{(0)}$ cada vez que se incrementa el nivel. El «objetivo siguiente», es decir, las diferencias $d_i = a_i^{(0)} - a_i^{(i-1)}$ se calculan también en ese momento. El algoritmo se apoya en el hecho de que los d_i resultantes son decrecientes a medida que aumenta i (escalera decreciente en la Fig. 2.16). (Obsérvese que la excepción se presenta en la transición del nivel 0 al nivel 1; este algoritmo, por tanto, debe utilizarse a partir de este nivel 1). *selecinta* termina disminuyendo d_j en 1; esta operación equivale a la sustitución de un tramo ficticio en la cinta j por uno real.

```
procedure selecinta;
  var i: nocinta; z: integer;
begin
  if d[j] < d[j + 1] then j := j + 1 else
  begin if d[j] = 0 then
    begin nivel := nivel + 1; z := a[1];
      for i := 1 to n - 1 do
        begin d[i] := z + a[i + 1] - a[i]; a[i] := z + a[i + 1]
        end
      end;
      j := 1
    end;
    d[j] := d[j] - 1
  end
end;
```

(2.46)

Suponiendo que se dispone de una rutina para copiar un tramo desde el fichero origen $f0$ sobre $f[j]$, puede formularse la fase de distribución inicial de la forma siguiente (siempre en el supuesto de que el fichero origen contiene un tramo como mínimo):

```
repeat selecinta; copitramo
until eof(f0)
```

(2.47)

Aquí, sin embargo, hay que detenerse un momento para recordar el efecto que se produjo en la distribución de tramos en el algoritmo de mezcla natural estudiado anteriormente: el hecho de que dos tramos que lleguen sucesivamente al mismo destino puedan llegar a constituir un tramo único puede dar lugar a que el número supuesto de tramos sea incorrecto. Puede ignorarse este efecto secundario si se diseña el algoritmo de ordenación de forma que su validez no dependa del número de tramos. En cambio, en el método polifásico es particularmente importante seguir la pista al número *exacto* de tramos en cada cinta. Por tanto, no puede pasarse por alto el efecto de estas mezclas por coincidencia.

Debido a esto no puede evitarse una complicación adicional en el algoritmo

de distribución. Es necesario retener las claves del último ítem del tramo final de cada cinta. Para esto se introduce una variable

ultimo: array [nocinta] of integer

El paso siguiente para describir el algoritmo de distribución podría ser

```
repeat selecinta;
  if ultimo[j] ≤ f0↑ . clave then
    «continuar el tramo antiguo»
    copitramo; ultimo[j] := f0↑ . clave
  until eof(f0)
```

(2.48)

Evidentemente, el error en este planteamiento está en olvidar que *ultimo[j]* solamente tiene definido su valor una vez copiado el primer tramo! Una solución inadecuada es distribuir primero un tramo sobre cada una de las $n - 1$ cintas sin inspeccionar *ultimo[j]*. Los siguientes tramos se distribuyen según (2.49)

```
while ¬eof(f0) do
begin selecinta;
  if ultimo[j] ≤ f0↑ . clave then
    begin {continuar el tramo antiguo}
      copitramo;
      if eof(f0) then d[j] := d[j] + 1 else copitramo
    end
    else copitramo
end
```

(2.49)

Aquí la asignación a *ultimo[j]* se supone incluida en el procedimiento *copitramo*.

A partir de este momento se está en situación de emprender la construcción del algoritmo principal del método de mezcla polifásico. Su estructura principal es similar a la del método de mezcla múltiple: un ciclo exterior que mezcla tramos hasta que se agotan los ficheros de las cintas origen, un ciclo interior que mezcla un tramo de cada cinta origen, y un ciclo interior a este último que selecciona la clave inicial y transmite el ítem correspondiente al fichero destino. Las principales diferencias del método polifásico respecto del de mezcla múltiple son las siguientes:

1. En cada pasada hay una sola cinta de destino en vez de $n/2$.
2. En lugar de intercambiar de función $n/2$ cintas origen y $n/2$ cintas destino en cada pasada, las cintas se cambian de origen a destino *rotativamente*. Esto se consigue empleando una tabla *c* de correspondencia de índices.
3. El número de cintas de entrada varía según el tramo que se procesa; se determina al principio del proceso de cada tramo, a partir de los contadores d_i de tramos ficticios. Si $d_i > 0$ para todos los i , hay que «pseudomezclar»

$n - 1$ tramos ficticios en un tramo ficticio, incrementando simplemente el contador d_n de la cinta de salida. En otro caso se mezcla un tramo de cada cinta que verifica $d_i = 0$, y se disminuye d_i en una unidad en todas las demás, indicándose con ello que se ha eliminado un tramo ficticio en cada una. El número de cintas de entrada que intervienen en un proceso de mezcla de tramos se denomina k .

4. Es imposible deducir la terminación de una fase por la aparición de la marca de fin de fichero en la cinta $n - 1$ -ésima, ya que pueden ser precisas más mezclas que utilicen tramos ficticios de esta cinta. En lugar de ello, el número de tramos teóricamente necesario se determina a partir de las coeficientes a_i . Los coeficientes $a_i^{(l)}$ se calcularon en la fase de distribución; ahora pueden recalcularse «hacia atrás».

El programa principal de ordenación por el método polifásico puede formularse ahora de acuerdo con estas reglas, suponiendo que las $n - 1$ cintas que tienen tramos iniciales están ya rebobinadas y que la tabla de correspondencia de cintas se inicia con $c_i = i$.

```
repeat {mezclar desde c[1] ... c[n - 1] hasta c[n]}
  z := a[n - 1]; d[n] := 0; rewrite(f[c[n]]);
  repeat k := 0; {mezclar un tramo}
    {determinar el numero k de cintas activas}
    for i := 1 to n - 1 do
      if d[i] > 0 then d[i] := d[i] - 1 else
        begin k := k + 1; cd[k] := c[i]
        end;
      if k = 0 then d[n] := d[n] + 1 else
        «mezclar un tramo real desde c[1] ... c[k]»;
    z := z - 1
  until z = 0;
  reset(f[c[n]]);
  «rotar las cintas en la tabla c; calcular las a[i] del nivel siguiente»;
  rewrite(f[c[n]]); nivel := nivel - 1
until nivel = 0;
{el resultado ordenado está en c[1]}
```

(2.50)

La propia operación de mezcla es casi idéntica a la del método múltiple; sólo se diferencian en que el algoritmo de eliminación de cintas es, en cierta forma, más simple. La programación del proceso de rotación de la tabla de correspondencia de índices de cintas y los contadores d_i (así como el recálculo de los coeficientes a_i) es inmediata y puede verse detallada en el Programa 2.16 que representa el algoritmo polifásico en su totalidad.

Programa 2.16. Ordenación por método polifásico.

```

program polisort (output);
{método polifásico con n cintas}
const n = 6; {no. de cintas}
type item = record
  clave: integer
end;
cinta = file of item;
nocinta = 1 .. n;
var long, alea: integer; {usados para generar el fichero}
fdc: boolean;
buf: item;
f0: cinta; {f0 es la cinta de entrada con numeros aleatorios}
f: array [1 .. n] of cinta;
procedure listar (var f: cinta; n: nocinta);
  var z: integer;
begin z := 0;
  writeln ('CINTA', n: 2);
  while ¬eof(f) do
    begin read(f, buf); write(output, buf .clave: 5); z := z + 1;
    if z = 25 then
      begin writeln (output); z := 0
      end
    end;
  if z ≠ 0 then writeln (output); reset(f)
end {listar};

procedure polifase;
  var i, j, mx, cn: nocinta;
  k, nivel: integer;
  a, d: array [nocinta] of integer;
  {a[j] = numero ideal de tramos en cinta j}
  {d[j] = numero de tramos ficticios en cinta j}
  dn, x, min, z: integer;
  ultimo: array [nocinta] of integer;
  {ultimo[j] = clave del ultimo item de la cinta j}
  c, cd: array [nocinta] of nocinta;
  {tablas de correspondencia de numeros de cintas}

procedure selecinta;
  var i: nocinta; z: integer;
begin
  if d[j] < d[j + 1] then j := j + 1 else
  begin if d[j] = 0 then

```

```

begin nivel := nivel + 1; z := a[1];
  for i := 1 to n - 1 do
    begin d[i] := z + a[i + 1] - a[i]; a[i] := z + a[i + 1]
    end
  end;
  j := 1
end;
d[j] := d[j] - 1
end;

procedure copitramo;
begin {copiar un tramo desde f0 a la cinta j}
  repeat read (f0, buf); write(f[j], buf);
  until eof(f0) ∨ (buf .clave > f0↑ .clave);
  ultimo[j] := buf .clave
end;

begin {distribuir los tramos iniciales}
  for i := 1 to n - 1 do
    begin a[i] := 1; d[i] := 1; rewrite(f[i])
    end;
  nivel := 1; j := 1; a[n] := 0; d[n] := 0;
  repeat selecinta; copitramo
  until eof(f0) ∨ (j = n - 1);
  while ¬eof(f) do
    begin selecinta;
    if ultimo[j] ≤ f0↑ .clave then
      begin {continuar el tramo antiguo}
        copitramo;
        if eof(f0) then d[j] := d[j] + 1 else copitramo
      end
    else copitramo
    end;
  for i := 1 to n - 1 do reset(f[i]);
  for i := 1 to n do c[i] := i;
  repeat {mezclar desde c[1] ... c[n - 1] hasta c[n]}
    z := a[n - 1]; d[n] := 0; rewrite(f[t[n]]);
    repeat k := 0; {mezclar un tramo}
      for i := 1 to n - 1 do
        if d[i] > 0 then d[i] := d[i] - 1 else
          begin k := k + 1; cd[k] := c[i]
          end;
    end;

```

Programa 2.16. (Continuación)

```

if  $k = 0$  then  $d[n] := d[n] + 1$  else
begin {mezclar un tramo real desde  $c[1] \dots c[k]$ }
repeat  $i := 1$ ;  $mx := 1$ ;
   $min := f[cd[1]] \uparrow .clave$ ;
  while  $i < k$  do
    begin  $i := i + 1$ ;  $x := f[cd[i]] \uparrow .clave$ ;
      if  $x < min$  then
        begin  $min := x$ ;  $mx := i$ 
        end
      end;
    { $cd[mx]$  contiene el elemento mínimo: moverlo a  $c[n]$ }
    read(f[cd[mx]], buf); fdc := eof(f[cd[mx]]);
    write(f[c[n]], buf);
    if (buf .clave > f[cd[mx]] \uparrow .clave)  $\vee$  fdc then
      begin {prescindir de esta cinta}
        cd[mx] := cd[k];  $k := k - 1$ 
      end
    end
  until  $k = 0$ 
end;
 $z := z - 1$ 
until  $z = 0$ ;
reset(f[c[n]]); listar(f[c[n]], t[n]): {rotar las cintas}
cn := c[n]; dn := d[n];  $z := a[n - 1]$ ;
for  $i := n$  downto 2 do
  begin  $c[i] := c[i - 1]$ ;  $d[i] := d[i - 1]$ ;  $a[i] := a[i - 1] - z$ 
  end;
 $c[1] := cn$ ;  $d[1] := dn$ ;  $a[1] := z$ ;
{el resultado ordenado está en  $c[1]$ }
listar(f[c[1]], c[1]); nivel := nivel - 1
until nivel = 0;
end {polifase};
begin {generar el fichero aleatorio}
long := 200; alea := 7789;
repeat alea := (131071 * alea) mod 2147483647;
  buf .clave := alea div 2147484; write(f0, buf); long := long - 1
until long = 0;
reset(f0); listar(f0, 1);
polifase
end.

```

Programa 2.16. (Continuación)

2.3.5. Distribución de los tramos iniciales

El desarrollo de los sofisticados programas de ordenación secuencial descritos anteriormente es una consecuencia de que los otros métodos más simples que operan sobre arrays no son aplicables, por no poder contar con una memoria de acceso aleatorio capaz de almacenar todos los datos a ordenar. Esta capacidad de memoria no está disponible a menudo; por ello es preciso emplear dispositivos de almacenamiento secuencial suficientemente grandes, como son las cintas. Se observa que los métodos de ordenación secuencial desarrollados hasta ahora, prácticamente, apenas necesitan memoria aparte del espacio para los buffers de los ficheros y, desde luego, para el propio programa. Sin embargo, en la realidad, incluso los pequeños computadores disponen de una memoria de acceso aleatorio que casi siempre es superior a la que necesitan los programas desarrollados hasta ahora. Es injustificable no utilizar esta capacidad de memoria en forma óptima.

La solución para ello es *combinar* las técnicas de ordenación de ficheros y arrays. En particular, puede utilizarse una versión adaptada de ordenación de arrays en la fase de distribución inicial de tramos siempre que estos tramos tengan una longitud de un tamaño aproximado al de la memoria principal disponible. Está claro que las pasadas de mezcla subsiguientes no pueden mejorarse mediante métodos de ordenación de arrays ya que los tramos implicados en esas pasadas aumentan de longitud y por ello superan la capacidad disponible en memoria principal. En consecuencia, puede concentrarse la atención en la mejora del algoritmo que genera los tramos iniciales.

Naturalmente, hay que limitar la búsqueda de métodos de ordenación de arrays a los de tipo logarítmico. El más adecuado de éstos es el método arboriforme o del montículo (ver apartado 2.2.5). El montículo puede considerarse como un túnel a través del que deben pasar todos los componentes del fichero, unos más deprisa y otros más despacio. La clave menor se toma directamente de la cumbre del montículo y el proceso de sustitución de ésta es muy eficiente. La acción de pasar un componente desde la cinta de entrada f_0 , a través de un montículo h que está lleno, a una cinta de salida $f[j]$, puede describirse sencillamente de la forma siguiente:

$$\begin{aligned}
&\text{write}(f[j], h[1]); \\
&\text{read}(f_0, h[1]); \\
&\text{criba}(1, n)
\end{aligned} \tag{2.51}$$

«Criba» es el proceso descrito en 2.25 para hacer que se hunda el componente recién insertado $h[1]$ hasta su sitio adecuado. Obsérvese que $h[1]$ es el menor ítem del montículo. En la Fig. 2.17 se muestra un ejemplo

El programa resulta considerablemente más complejo debido a:

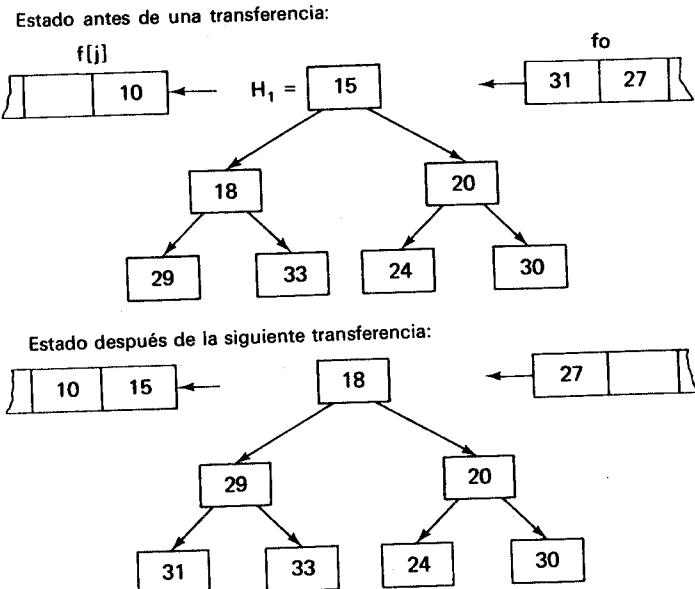


Fig. 2.17. Criba por hundimiento de un ítem en el montículo.

1. El montículo h está vacío inicialmente, y debe llenarse al principio.
2. Hacia el final del proceso, el montículo está lleno parcialmente y, al final, debe vaciarse.
3. Es preciso controlar el comienzo de nuevos tramos para cambiar el índice j de la cinta de salida en el momento adecuado.

Antes de proceder a la definición del proceso se declaran formalmente las variables que evidentemente están en juego:

```

var f0: cinta;
f: array [nocinta] of cinta;
h: array [1 .. m] of item;
iz, de: integer
  
```

(2.52)

m es el tamaño del montículo h . Se utiliza la constante $m2$ para designar $m/2$; iz y de son índices en h . El proceso de hundimiento a través del montículo puede dividirse en cinco partes distintas:

1. Lectura de los $m2$ primeros ítems de $f0$ y colocación en la mitad superior del montículo en donde no es necesario que estén ordenados.
2. Lectura de otros $m2$ ítems y situación de los mismos en la mitad inferior del montículo dejando hundirse cada ítem hasta su posición apropiada (constucción del montículo).

3. Asignar a iz el valor m y repetir el paso siguiente con los restantes ítems de $f0$: Pasar $h[1]$ a la cinta de salida apropiada. Si su clase es menor o igual que la del siguiente ítem de la cinta de entrada, este último ítem pertenece al mismo tramo y puede dejársele hundir a través del montículo hasta su posición adecuada. En caso contrario se reduce el tamaño del montículo y se sitúa el nuevo ítem en un segundo montículo «superior» que se construye para contener el tramo siguiente. La línea fronteriza entre ambos montículos se indica con el índice iz . Así, el montículo «inferior» que está en curso de proceso contiene los ítems $h[1], \dots, h[iz]$ y el «superior», siguiente a operar, contiene $h[iz + 1] \dots h[m]$. Si $iz = 0$ se cambia de cinta de salida y se asigna a iz el valor m .
4. Cuando el fichero origen está agotado se asigna a de el valor m ; se vacía la parte inferior para terminar el tramo en curso, y al mismo tiempo se construye la parte superior que se almacena en las posiciones $h[iz + 1] \dots h[de]$.
5. El último tramo se genera a partir de los ítems que quedan en el montículo.

Con este planteamiento se está en situación de describir las cinco partes como un programa completo, que llame a un procedimiento *selecinta* cada vez que se detecta el final de un tramo y haya que realizar alguna acción para alterar el índice de la cinta de salida. En el Programa 2.17 se ha utilizado en su lugar una rutina ficticia que se limita a contar el número de tramos generados. Todos los elementos se escriben sobre la cinta $f1$.

Si ahora se intenta integrar este programa con el del método polifásico, por ejemplo, aparecería una dificultad seria. Esta se presenta como consecuencia de las siguientes circunstancias: el programa de ordenación contiene en su parte inicial una rutina bastante complicada para cambio de cintas y se basa en la posibilidad de disponer de un procedimiento *copitramo* que transmite exactamente un tramo a la cinta seleccionada. El programa que aplica el método del montículo, por otro lado, es una rutina compleja basada en la disponibilidad de un procedimiento cerrado denominado *selecinta* que, simplemente, selecciona una nueva cinta. No habría problema si en uno (o en ambos) programas se llamara al procedimiento deseado en un solo sitio; pero en lugar de esto se llama a ambos en varios sitios.

Esta situación se refleja mejor utilizando lo que se denomina una *corrutina*, cuyo uso es adecuado cuando coexisten varios procesos. El caso más típicamente representativo es la combinación de un proceso que produce un flujo de información en distintas entidades, y otro proceso que consume este flujo. Esta relación productor/consumidor puede expresarse en base a dos corrutinas. Una de ellas puede bien ser el propio programa.

La corrutina puede considerarse como un procedimiento o subrutina que contiene uno o varios puntos de discontinuidad o ruptura (*«breakpoints»*). Si se encuentra uno de estos puntos el control vuelve al programa que había llamado a la corrutina. Cuando vuelve a llamarse a la corrutina, ésta continúa su ejecu-

Programa 2.17. Distribución de tramos iniciales a través de un montículo.

```

program distribucion(f0, f1, output);
{distribucion inicial de tramos por el metodo del monticulo}
const m = 30; m2 = 15; {tamaño del monticulo}
type item = record
    clave: integer
end;
cinta = file of item;
indice = 0 .. m;
var iz, de: indice;
f0, f1: cinta;
conta: integer; {contador de tramos}
h: array [1 .. m] of item; {monticulo}

procedure selecinta;
begin conta := conta + 1;
{ficticio; contar el número de tramos distribuidos}
end {selecinta};

procedure criba(iz, de: indice);
label 13;
var i, j: integer; x: item;
begin i := iz; j := 2 * i; x := h[i];
while j ≤ de do
begin if j < de then
    if h[j].clave > h[j + 1].clave then j := j + 1;
    if x.clave ≤ h[j].clave then goto 13;
    h[i] := h[j]; i := j; j := 2 * i
end;
13: h[i] := x
end;

begin {crear los tramos iniciales ordenando por el metodo del monticulo}
conta := 0; reset(f0); rewrite(f1);
selecinta;
{paso 1: llenar la mitad superior del monticulo h}
iz := m;
repeat read(f0, h[iz]); iz := iz - 1
until iz = m2;
{paso 2: llenar la mitad inferior del monticulo h}
repeat read(f0, h[iz]); criba(iz, m); iz := iz - 1
until iz = 0;
{paso 3: pasar los tramos por todo el monticulo}
iz := m;

```

```

while ¬eof(f0) do
begin write(f1, h[1]);
if h[1].clave ≤ f0↑.clave then
begin {el nuevo registro pertenece al mismo tramo}
read(f0, h[1]); criba(1, iz);
end else
begin {el nuevo registro pertenece al tramo siguiente}
h[1] := h[iz]; criba(1, iz - 1);
read(f0, h[iz]); if iz ≤ m2 then criba(iz, m); iz := iz - 1;
if iz = 0 then
begin {el monticulo esta completo; se empieza con un nuevo tramo}
iz := m; selecinta;
end
end;
end;
{paso 4: vaciar la parte inferior del monticulo}
de := m;
repeat write(f1, h[1]);
h[1] := h[iz]; criba(1, iz - 1);
h[iz] := h[de]; de := de - 1;
if iz ≤ m2 then criba(l, r); l := l - 1
until iz = 0;
{paso 5: vaciar la parte superior del monticulo; generar el ultimo tramo}
selecinta;
while de > 0 do
begin write(f1, h[1]);
h[1] := h[de]; criba(1, de); de := de - 1
end;
writeln (conta)
end.

```

Programa 2.17. (Continuación)

ción a partir del punto donde se interrumpió. En el caso que se estudia puede considerarse el método polifásico como programa principal que llama a *copitramo*, el cual se formula como corrutina. Está formado por el cuerpo principal del Programa 2.17, en el que cada llamada a *selecinta* representa un punto de ruptura. La comprobación de fin de fichero tendría que reemplazarse sistemáticamente por una pregunta de si la corrutina ha llegado o no a su final. Una formulación lógica sería *fdc* (*copitramo*) en lugar de *eof(f0)*.

Análisis y conclusiones. ¿Qué rendimiento puede esperarse de un método mixto constituido por el polifásico con distribución inicial de tramos por el mé-

todo del montículo? Se estudia inicialmente la mejora que puede esperarse al introducir el método del montículo.

La longitud media esperada de cada tramo en una sucesión con claves distribuidas aleatoriamente es 2. ¿Cuál es su longitud una vez que la sucesión ha pasado a través de un montículo de tamaño m ? A primera vista podría decirse que m . Sin embargo, afortunadamente, el resultado del análisis probabilístico es mucho mejor, concretamente, $2m$ (ver Knuth Vol. 3, pág. 254). Por ello, el factor de mejora esperado es m .

Puede tenerse una estimación del rendimiento del método polifásico a partir de la Tabla 2.15, que indica el máximo número de tramos iniciales que pueden ordenarse con un número prefijado de pasadas (niveles) y un número dado de cintas n . Como ejemplo, con $n = 6$ cintas y un montículo de tamaño $m = 100$ puede ordenarse un fichero de hasta 165680100 tramos iniciales en un número de pasadas inferior a 20. Este rendimiento es notable.

Volviendo de nuevo a la combinación del método polifásico y el del montículo, es inevitable sentirse intrigado por la complejidad de este programa. Al fin y al cabo, éste realiza la misma tarea, fácil de enunciar, de reordenar un conjunto de ítems, tal como lo hace cualquiera de los pequeños programas que se basan en los principios de ordenación directa de arrays. La lección a extraer de todo el capítulo puede resumirse en los dos aspectos siguientes:

1. La íntima conexión que hay entre el algoritmo y la estructura de datos subyacente y, particularmente, la influencia de esta última en el primero.
2. La sofisticación con que puede mejorarse el funcionamiento de un programa, incluso cuando la estructura disponible de sus datos (fichero secuencial en lugar de array) es poco adecuada para esta tarea.

E J E R C I C I O S

- 2.1. ¿Cuáles de los algoritmos dados por los Programas 2.1 hasta 2.6, 2.8, 2.10 y 2.13 son métodos estables de ordenación?
- 2.2. El Programa 2.2, ¿seguiría funcionando correctamente si en la cláusula **while** se sustituyera $iz \leq de$ por $iz < de$? ¿Continuaría siendo correcto si las instrucciones $de := m - 1$ y $iz := m + 1$ se simplificaran a $de := m$ y $iz := m$? Si no fuera correcto, encontrar conjuntos de valores $a_1 \dots a_n$ en los que fallara el programa modificado.
- 2.3. Programar y medir el tiempo de ejecución de tres métodos directos de ordenación en el computador del lector, y encontrar los coeficientes que deben afectar a C y M para producir estimaciones realistas del tiempo de ejecución.
- 2.4. Ensayar el Programa 2.8 del método del montículo con varias sucesiones de entrada aleatorias y determinar el número medio de veces que se ejecuta la instrucción **goto** 13. Dado que este número es relativamente pequeño es interesante la pregunta siguiente:

¿Hay alguna forma de sacar la comprobación

$$x . clave \geq a[j] . clave$$

fuera del bucle «while»?

- 2.5. Considérese la siguiente versión «obvia» del programa de partición 2.9:

```
i := 1; j := n;
x := a[(n + 1) div 2] . clave;
repeat
    while a[i] . clave < x do i := i + 1;
    while x < a[j] . clave do j := j - 1;
    w := a[i]; a[i] := a[j]; a[j] := w
until i > j
```

Encontrar conjuntos de valores $a_1 \dots a_n$ para los que falla esta versión.

- 2.6. Escribir un programa que combine el método rápido y el de la burbuja de la forma siguiente: utilizar el método rápido para obtener particiones (no ordenadas) de longitud m ($1 \leq m \leq n$); a continuación utilizar el método de la burbuja para completar la tarea. Obsérvese que este último puede inspeccionar el array total de n elementos, y, por ello, minimizar el esfuerzo de administración de datos. Encontrar el valor de m que minimiza el tiempo total de ordenación.

Nota: Evidentemente, el valor óptimo de m será muy pequeño. Por ello puede compensar hacer, por el método de la burbuja, exactamente $m - 1$ recorridos sobre el array, en vez de incluir una última pasada para establecer el hecho de que no son necesarios nuevos intercambios.

- 2.7. Realizar el mismo experimento que en 2.6 con el método de selección directa en vez del método de la burbuja. Naturalmente, el método de selección no puede recorrer todo el array; por ello el esfuerzo esperado de manejo de índices será algo mayor.
- 2.8. Escribir un algoritmo recursivo para el método rápido siguiendo la norma de comenzar la ordenación de la partición más corta antes que la más larga. Realizar la primera tarea mediante una instrucción iterativa y la segunda mediante una llamada recursiva (por tanto, el programa resultante tendrá una llamada recursiva, en vez de las dos del Programa 2.10, y ninguna del Programa 2.11).
- 2.9. Encontrar una permutación de las claves $1, 2, \dots, n$ con la que el método rápido funcione de la peor (mejor) forma posible ($n = 5, 6, 8$).
- 2.10. Construir un programa de mezcla natural parecido al programa de mezcla directa 2.13, que opere sobre un array de doble tamaño desde ambos extremos hacia el interior; comparar su rendimiento con el del Programa 2.13.
- 2.11. Obsérvese que en un proceso de mezcla natural (doble) no se selecciona a ciegas el menor valor entre las claves posibles. En lugar de ello, al encontrar el final de un tramo, se copia simplemente el final del otro tramo sobre la sucesión de salida. Por ejemplo al mezclar

2, 4, 5, 1, 2, ...

3, 6, 8, 9, 7, ...

132 ORDENACIÓN

se produce la secuencia

2, 3, 4, 5, 6, 8, 9, 1, 2, ...

en lugar de

2, 3, 4, 5, 1, 2, 6, 8, 9, ...

que parece estar mejor ordenada. ¿Cuál es la razón de esta estrategia?

- 2.12. ¿Para qué sirve la variable *cd* en el Programa 2.15? ¿En qué circunstancias se ejecuta la instrucción

```
begin rewrite (f[cd[mx]]); ...
```

y cuándo se ejecuta

```
begin cx := cd[mx]; ...?
```

- 2.13. ¿Por qué es necesaria la variable *ultimo* en el Programa 2.16 del método polifásico, y, en cambio, no es necesaria en el Programa 2.15?

- 2.14. Un método de ordenación semejante al polifásico es el denominado de *mezcla en cascada* [2.1 y 2.9]. Utiliza un esquema de mezcla diferente. Dadas, por ejemplo, seis cintas C_1, \dots, C_6 , el método de mezcla en cascada, que también comienza con una «distribución perfecta» de tramos sobre C_1, \dots, C_5 realiza una mezcla múltiple desde C_1, \dots, C_5 sobre C_6 hasta vaciar C_5 , a continuación (sin tocar C_6) una mezcla desde C_1, \dots, C_4 sobre C_5 , luego otra desde C_1, C_2, C_3 sobre C_4 , otra desde C_1, C_2 sobre C_3 y, finalmente, una operación de copia de C_1 sobre C_2 . La siguiente pasada opera de la misma forma empezando con una mezcla desde cinco cintas hacia C_1 , y así sucesivamente. Aunque este esquema parece inferior al polifásico debido a que a veces, deja algunas cintas sin usar, e incluye operaciones de simple copiado, sorprendentemente supera al método polifásico en casos de ficheros (muy) voluminosos y de seis o más cintas. Escribir un programa bien estructurado para el método de mezcla en cascada.

REFE R E N C I A S

- 2-1. BETZ, B. K. y CARTER, *ACM National Conf.*, **14**, (1959), Artículo 14.
- 2-2. FLOYD, R. W., «Treesort» (Algoritmos 113 y 243), *Comm. ACM*, **5**, No. 8 (1962), 434, y *Comm. ACM*, **7**, No. 12 (1964), 701.
- 2-3. GILSTAD, R. L., «Polyphase Merge Sorting...An Advanced Technique», *Proc. AFIPS Eastern Jt. Comp. Conf.*, **18**, (1960), 143-48.
- 2-4. HOARE, C. A. R., «Proof of a Program: FIND», *Comm. ACM*, **13**, No. 1 (1970), 39-45.

- 2-5., «Proof of a Recursive Program: Quicksort», *Comp. J.*, **14**, No. 4 (1971), 391-95.
- 2-6., «Quicksort», *Comp. J.*, **5**, No. 1 (1962), 10-15.
- 2-7. KNUTH, D. E., *The Art of Computer Programming*, Vol. 3 (Reading, Mass: Addison-Wesley, 1973).
- 2-8., *The Art of Computer Programming*, **3**, pp. 86-95.
- 2-9., *The Art of Computer Programming*, **3**, p. 289.
- 2-10. LORIN, H., «A Guided Bibliography to Sorting», *IBM Syst. J.*, **10**, No. 3 (1971), 244-54.
- 2-11. SHELL, D. L., «A Highspeed Sorting Procedure», *Comm. ACM*, **2**, No. 7 (1959), 30-32.
- 2-12. SINGLETON, R. C., «An Efficient Algorithm for Sorting with Minimal Storage» (Algoritmo 347), *Comm. ACM*, **12**, No. 3 (1969), 185.
- 2-13. VAN EMDEN, M. H., «Increasing the Efficiency of Quicksort» (Algoritmo 402), *Comm. ACM*, **13**, No. 9 (1970), 563-66, 693.
- 2-14. WILLIAMS, J. W. J., «Heapsort» (Algoritmo 232), *Comm. ACM*, **7**, No. 6 (1964), 347-48.

3**ALGORITMOS RECURSIVOS****3.1. INTRODUCCION**

Se dice que un objeto es *recursivo* si forma parte de sí mismo o se define en función de sí mismo. La recursión aparece no sólo en matemáticas, sino también en la vida diaria. ¿Quién no ha visto alguna vez un anuncio que se contiene a sí mismo?



Fig. 3.1. Imagen recursiva.

La recursión es un medio particularmente poderoso en las definiciones matemáticas. Algunos ejemplos conocidos de ello son los números naturales, las estructuras árbol y ciertas funciones:

1. Números naturales:
 - (a) 1 es un número natural.
 - (b) el siguiente de un número natural es un número natural.
2. Estructuras árbol
 - (a) \circ es un árbol (denominado el árbol vacío).
 - (b) si t_1 y t_2 son árboles, entonces



es un árbol (dibujado boca abajo)

3. La función factorial, $n!$ (para enteros no negativos):
 - (a) $0! = 1$
 - (b) si $n > 0$ entonces $n! = n \cdot (n - 1)!$

La potencia de la recursión reside evidentemente en la posibilidad de definir un número infinito de objetos mediante un enunciado finito. De igual forma, un número infinito de operaciones de cálculo puede describirse mediante un programa recursivo finito, incluso si este programa no contiene repeticiones explícitas. De todas formas, los algoritmos recursivos son apropiados principalmente cuando el problema a resolver, o la función a calcular, o la estructura de datos a procesar, están ya definidos en forma recursiva. En general, un programa recursivo P puede expresarse como una composición \mathcal{G} de instrucciones básicas S_i (que no contienen a P) y el propio P

$$P \equiv \mathcal{G}[S_i, P] \quad (3.1)$$

El instrumento necesario y suficiente para expresar los programas recursivamente es el *procedimiento* («procedure») o subrutina, ya que permite dar un nombre a una instrucción por el cual ésta puede ser llamada. Si un procedimiento P contiene una referencia explícita a sí mismo se dice que es *directamente recursivo*; si P contiene una referencia a otro procedimiento Q que, a su vez, contiene una referencia (directa o indirecta) a P , se dice que P es *indirectamente recursivo*. El uso de la recursión puede, por ello, no hacerse patente de forma inmediata en el texto del programa.

Es corriente asociar un conjunto de objetos locales a un procedimiento, es decir un conjunto de variables, constantes, definiciones de tipo, y procedimientos, que se definen localmente para este procedimiento y no existen, o carecen de significado, fuera del procedimiento. Cada vez que se activa recursivamente un procedimiento de esta clase, se crea un nuevo conjunto de variables locales ligadas al procedimiento. Aunque tienen los mismos nombres que los elementos correspondientes en el conjunto local de la anterior llamada al procedimiento, sus valores son distintos, y cualquier conflicto de nombres se evita mediante las reglas de campo de validez de los identificadores: éstos se refieren siempre al conjunto de variables creado más recientemente. Esta misma regla vale para los

parámetros tipo procedimiento que, por definición, se encuentran ligados al procedimiento.

Lo mismo que las instrucciones repetitivas, los procedimientos recursivos pueden realizar procesos que no terminan, y de aquí la necesidad de tener en cuenta el problema de la *terminación*. Para ello, evidentemente, es fundamental que la llamada recursiva a un procedimiento P dependa de una condición B que en algún momento no se satisfaga. Por lo tanto, el esquema de un algoritmo recursivo puede expresarse más precisamente en la forma

$$P \equiv \text{if } B \text{ then } \varnothing[S_i, P] \quad (3.2)$$

o bien

$$P \equiv \varnothing[S_i, \text{if } B \text{ then } P] \quad (3.3)$$

La técnica básica para demostrar que una repetición termina es definir una función $f(x)$ (siendo x el conjunto de variables del programa), tal que $f(x) \leq 0$ implique la condición de terminación (de las cláusulas «while» o «repeat»), y demostrar que el valor de $f(x)$ disminuye en cada repetición. De igual forma, puede demostrarse la terminación de un programa recursivo comprobando que el valor de $f(x)$ disminuye en cada ejecución de P . Una manera particularmente evidente de asegurar la terminación es asociar a P un parámetro (constante), por ejemplo n , y llamar a P recursivamente con $n - 1$ como parámetro (constante). Si se sustituye la condición B por $n > 0$, se garantiza la terminación. Esto se puede expresar mediante los siguientes esquemas de programa:

$$P(n) \equiv \text{if } n > 0 \text{ then } \varnothing[S_i, P(n-1)] \quad (3.4)$$

$$P(n) \equiv \varnothing[S_i, \text{if } n > 0 \text{ then } P(n-1)] \quad (3.5)$$

En las aplicaciones prácticas es imperativo demostrar que el nivel máximo de recursión es no solo finito, sino realmente pequeño. La razón es que se necesita cierta cantidad de memoria para almacenar las variables correspondientes a cada activación recursiva de un procedimiento P . Además de estas variables locales, hay que almacenar el estado en curso del proceso de cálculo para recuperarlo cuando se acabe la nueva activación de P , y haya que reanudar la antigua. Se ha presentado ya esta situación al desarrollar el procedimiento rápido de ordenación («Quicksort») en el Cap. 2. Se comprobó que al diseñar «ingenuamente» el programa a partir de una instrucción que subdividía los n ítems en dos particiones y de dos llamadas recursivas que ordenaban ambas particiones, el nivel de la recursión podía aproximarse en el caso más desfavorable, a n . Mediante un hábil replanteamiento de la situación fue posible limitar este nivel de recursión a $\log n$. La diferencia entre n y $\log n$ es suficiente para convertir un caso altamente inadecuado para la recursión en otro en el que ésta resulta perfectamente práctica.

3.2. CUANDO NO UTILIZAR LA RECURSIÓN

Los algoritmos recursivos son particularmente apropiados cuando el problema a resolver o los datos a tratar se definen en forma recursiva. Sin embargo, esto no significa que tales definiciones recursivas garanticen que la mejor forma de resolver el problema sea un algoritmo recursivo. De hecho, la explicación del concepto de algoritmo recursivo mediante ejemplos inapropiados ha sido la causa principal de que se haya creado una antipatía general hacia el uso de la recursión en programación, asociándose recursión a ineficacia.

A esto se ha unido el hecho de que el lenguaje FORTRAN, de utilización generalizada, prohíbe el uso recursivo de las subrutinas, y por tanto evita el diseño de soluciones recursivas, incluso cuando son apropiadas.

Los programas en que debe evitarse el uso de la recursión algorítmica pueden caracterizarse mediante un esquema que pone de relieve su forma de composición. El esquema es el (3.6) o, equivalentemente, el (3.7)

$$P \equiv \text{if } B \text{ then } (S; P) \quad (3.6)$$

$$P \equiv (S; \text{if } B \text{ then } P) \quad (3.7)$$

Estos esquemas son típicos de aquellos casos en que deben calcularse valores definidos en forma de relaciones de recurrencia simples. Obsérvese el conocido ejemplo de los números factoriales $f_i = i!$:

$$i = 0, 1, 2, 3, 4, 5, \dots \quad (3.8)$$

$$f_i = 1, 1, 2, 6, 24, 120, \dots$$

El factorial de cero se define explícitamente como $f_0 = 1$, mientras los factoriales siguientes se definen usualmente —en forma recursiva— a partir de su precedente:

$$f_{i+1} = (i + 1) \cdot f_i \quad (3.9)$$

Esta fórmula sugiere un algoritmo recursivo para el cálculo del factorial de n . Si se introducen las dos variables I, F para designar los valores i, f_i al nivel i de recursión, se ve que el cálculo necesario para obtener los números siguientes en la secuencia (3.8) debe ser

$$I := I + 1; \quad F := I * F \quad (3.10)$$

y, sustituyendo S por (3.10) en 3.6, se obtiene el programa recursivo

$P \equiv \text{if } I < n \text{ then } (I := I + 1; F := I * F; P)$ (3.11)
 $I := 0; F := 1; P$

La primera línea de (3.11) se expresa, según la notación convencional establecida, de la forma:

```
procedure P;
begin if  $I < n$  then
    begin  $I := I + 1; F := I * F; P$ 
    end
end
```

 (3.12)

Otra forma utilizada más frecuentemente, pero esencialmente equivalente, es la dada en (3.13). P es sustituido por un procedimiento del tipo llamado función, es decir un procedimiento al que se asocia explícitamente un valor resultado y que, por ello, puede utilizarse directamente para formar expresiones. De aquí que la variable F resulte superflua, y el papel de I sea incorporado por el parámetro explícito del procedimiento.

```
function F( $I$ : integer): integer;
begin if  $I > 0$  then  $F := I * F(I - 1)$ 
else  $F := 1$ 
end
```

 (3.13)

Está absolutamente claro que en este caso la recursión puede sustituirse por la simple iteración, en concreto, por el programa

```
 $I := 0; F := 1;$ 
while  $I < n$  do
    begin  $I := I + 1; F := I * F$ 
    end
```

 (3.14)

En general, los programas correspondientes a los esquemas genéricos (3.6) o (3.7) deberían transcribirse a uno acorde con el esquema (3.15)

$P \equiv (x := x_0; \text{while } B \text{ do } S)$ (3.15)

También hay esquemas recursivos más complicados que pueden y deben ponerse en forma iterativa. Un ejemplo de esto es el cálculo de los números de Fibonacci que se definen por la relación de recurrencia

$$\text{fib}_{n+1} = \text{fib}_n + \text{fib}_{n-1} \quad \text{para } n > 0 \quad (3.16)$$

y $\text{fib}_1 = 1$, $\text{fib}_0 = 0$. Un planteamiento directo y trivial conduce al programa

```
function Fib( $n$ : integer): integer;
begin if  $n = 0$  then Fib := 0 else
    if  $n = 1$  then Fib := 1 else
        Fib := Fib( $n - 1$ ) + Fib( $n - 2$ )
    end
```

 (3.17)

El cálculo de fib_n mediante una llamada a $\text{Fib}(n)$ da lugar a que este procedimiento función se active recursivamente. ¿Con qué frecuencia? Se observa que cada llamada con $n > 1$ produce otras 2 nuevas llamadas, es decir, el número total de llamadas crece exponencialmente (ver Fig. 3.2). Tal programa, evidentemente, no es práctico.

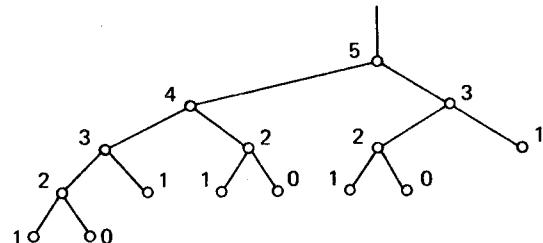


Fig. 3.2. Las 15 llamadas a $\text{Fib}(n)$ para $n = 5$.

Sin embargo, está claro que pueden calcularse los números de Fibonacci mediante un esquema iterativo que evita la duplicación del cálculo de los mismos valores utilizando como variables auxiliares $x = \text{fib}_i$ e $y = \text{fib}_{i-1}$.

```
{calcular  $x = \text{fib}_n$  para  $n > 0\}$ 
 $i := 1; x := 1; y := 0;$ 
while  $i < n$  do
    begin  $z := x; i := i + 1;$ 
           $x := x + y; y := z$ 
    end
```

 (3.18)

(Obsérvese que las tres asignaciones a x , y , z pueden expresarse mediante dos únicas asignaciones sin necesidad de la variable auxiliar z : $x := x + y$; $v := x - y$).

Por tanto, la lección a retener de lo anterior es que debe evitarse el uso de la recursión cuando haya una solución obvia por iteración.

Sin embargo, esto no debe llevar a huir de la recursión a cualquier precio. Como se demostrará en los apartados y capítulos siguientes, hay aplicaciones muy útiles de la recursión. El hecho de que existan implantadas aplicaciones de

procedimientos recursivos en máquinas esencialmente no recursivas, demuestra que, a efectos prácticos, todo programa recursivo puede transformarse en otro iterativo. Sin embargo, esto implica el manejo explícito de una «pila» de recursión, y estas operaciones pueden, a menudo, obscurecer la esencia de un programa, de forma que resulte difícil de comprender. Por tanto, los algoritmos que por su naturaleza son más recursivos que iterativos deben formularse como procedimientos recursivos. Para apreciar este punto, el lector puede comparar los Programas 2.10 y 2.11.

El resto de este capítulo se dedica al desarrollo de algunos programas recursivos en casos en que el uso de la recursión está justificado. También los Caps. 4 y 5 utilizan de manera importante la recursión en los casos en que las estructuras de datos subyacentes hacen aparecer como obvia y natural la elección de soluciones recursivas.

3.3. DOS EJEMPLOS DE PROGRAMAS RECURSIVOS

La atractiva forma gráfica que se muestra en la Fig. 3.5 está formada por la superposición de cinco curvas. Estas siguen un esquema regular y sugieren que deben poder dibujarse por un «plotter» controlado por computador. El objetivo a conseguir es descubrir el esquema recursivo con el que debe construirse el programa que haga el dibujo. Por inspección se observa que tres de las curvas superpuestas tienen la forma indicada en la Fig. 3.3; se designan por H_1 , H_2 y H_3 . Las

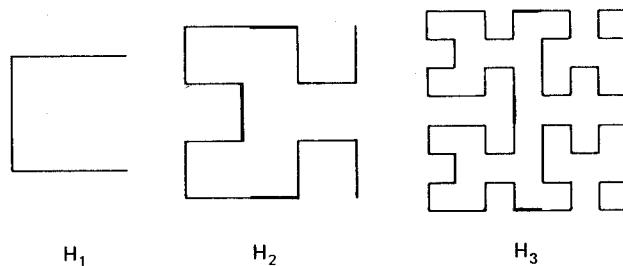


Fig. 3.3. Curvas de Hilbert de orden 1, 2 y 3.

figuras muestran que H_{i+1} se obtiene por composición de cuatro curvas de tipo H_i de tamaño mitad, giradas apropiadamente, y unidas por tres líneas. Obsérvese que H_1 puede considerarse formada por cuatro curvas de un tipo vacío H_0 conectadas por tres rectas. H_i se denomina *curva de Hilbert* de orden i , en honor a su inventor D. Hilbert (1891).

Supóngase que las herramientas básicas de que se dispone para dibujar son dos variables de coordenadas x e y , un procedimiento *setplot* (situar la pluma del «plotter» en las coordenadas x e y) y un procedimiento *plot* (que mueve la pluma de dibujo desde la situación actual a la posición indicada por x e y).

Como cada curva H_i está formada por cuatro copias de tamaño mitad de la

curva H_{i-1} es lógico expresar el procedimiento de dibujar H_i como compuesto de cuatro partes, cada una dibujando una H_{i-1} del tamaño apropiado, convenientemente girada. Si se denomina cada parte, respectivamente, por A , B , C y D , y las rutinas que dibujan las correspondientes líneas de interconexión se representan por flechas apuntando en la dirección correspondiente aparece el siguiente *esquema recursivo* (ver Fig. 3.3).

$$\begin{array}{l} \square A: D \leftarrow A \downarrow A \rightarrow B \\ \square B: C \uparrow B \rightarrow B \downarrow A \\ \square C: B \rightarrow C \uparrow C \leftarrow D \\ \square D: A \downarrow D \leftarrow D \uparrow C \end{array} \quad (3.19)$$

Si se designa la línea unitaria por h , el procedimiento correspondiente al esquema A se expresa inmediatamente utilizando activaciones recursivas de los procedimientos designados análogamente por B y D y del propio A .

```
procedure A(i: integer);
begin if i > 0 then
    begin D(i-1); x := x-h; plot;
          A(i-1); y := y-h; plot;
          A(i-1); x := x+h; plot;
          B(i-1)
    end
end
```

Este procedimiento se inicia por el programa principal una vez por cada curva de Hilbert a superponer. El programa principal determina el punto inicial de la curva, o sea, los valores iniciales de x e y , y el incremento unitario h . Se llama h_0 al ancho total de la página, que debe ser $h_0 = 2^k$ para algún $k \geq n$ (ver Fig. 3.4). El programa completo dibuja las n curvas de Hilbert $H_1 \dots H_n$ (ver el Programa 3.1 y la Fig. 3.5).

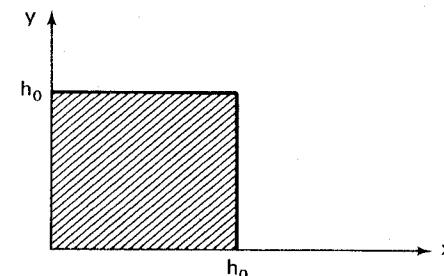


Fig. 3.4. El marco unitario.

Program 3.1. Curvas de Hilbert.

```

program Hilbert(pf,output);
{dibujar las curvas de Hilbert de ordenes 1 a n}
const n = 4; h0 = 512;
var i,h,x,y,x0,y0: integer;
    fd: file of integer; {fichero del dibujo}
procedure A(i: integer);
begin if i > 0 then
    begin D(i-1); x := x-h; plot;
        A(i-1); y := y-h; plot;
        A(i-1); x := x+h; plot;
        B(i-1)
    end
end ;
procedure B(i: integer);
begin if i > 0 then
    begin C(i-1); y := y+h; plot;
        B(i-1); x := x+h; plot;
        B(i-1); y := y-h; plot;
        A(i-1)
    end
end ;
procedure C(i: integer);
begin if i > 0 then
    begin B(i-1); x := x+h; plot;
        C(i-1); y := y+h; plot;
        C(i-1); x := x-h; plot;
        D(i-1)
    end
end ;
procedure D(i: integer);
begin if i > 0 then
    begin A(i-1); y := y-h; plot;
        D(i-1); x := x-h; plot;
        D(i-1); y := y+h; plot;
        C(i-1)
    end
end ;
begin startplot;
    i : = 0; h := h0; x0 : = h div 2; y0 : = x0;

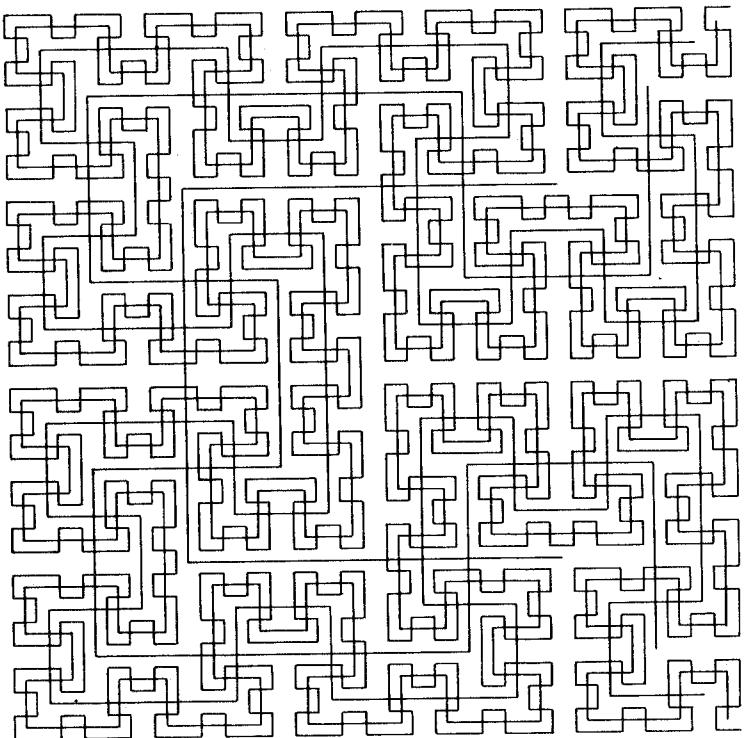
```

```

repeat {dibujar la curva de Hilbert de orden i}
    i := i+1; h := h div 2;
    x0 := x0 + (h div 2); y0 := y0 + (h div 2);
    x := x0; y := y0; setplot;
    A(i)
until i = n;
endplot
end.

```

Programa 3.1. (Continuación)

Fig. 3.5. Curvas de Hilbert $H_1 \dots H_5$.

Un ejemplo semejante, pero algo más complejo y más sofisticado desde el punto de vista estético, se muestra en la Fig. 3.7. Esta forma se obtiene también por superposición de varias curvas, dos de las cuales se exhiben en la Fig. 3.6. S_i se denomina la curva de Sierpinski de orden i . ¿Cuál es el esquema recursivo? Se siente uno tentado de identificar la «hoja» S_1 como bloque básico de construcción, posiblemente quitándole un borde. Pero esto no conduce a una solución. La diferencia principal entre las curvas de Hilbert y Sierpinski es que estas últi-

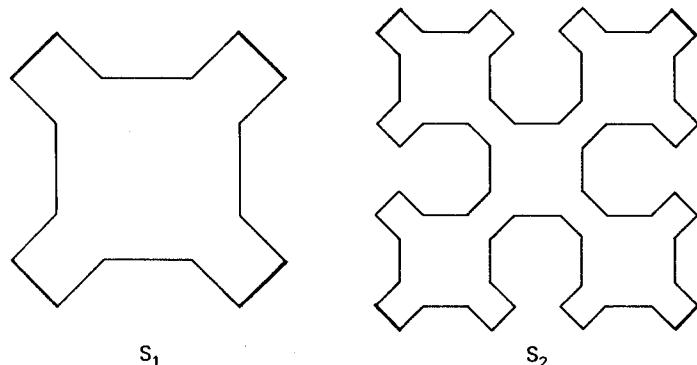
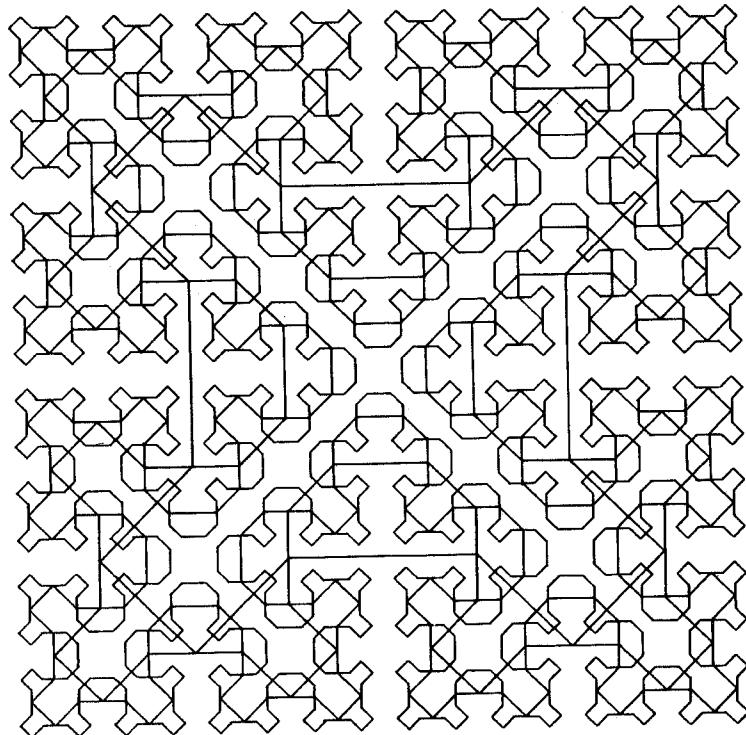


Fig. 3.6. Curvas de Sierpinski de órdenes 1 y 2.

Fig. 3.7. Curvas de Sierpinski $S_1 \dots S_4$.

mas son cerradas (sin discontinuidades). Esto implica que el esquema básico de recursión debe ser una curva abierta y que las cuatro partes deben estar conec-

tadas por líneas no pertenecientes al propio esquema recursivo. Realmente, estas líneas están formadas por cuatro rectas en las cuatro esquinas de los bordes, dibujadas en trazo grueso en la Fig. 3.6. Pueden considerarse como pertenecientes a una curva inicial *no vacía*, S_0 , que es un cuadrado apoyado en un vértice.

A partir de esto, el esquema recursivo se establece inmediatamente. Las formas básicas se denominan nuevamente A , B , C y D y las líneas de conexión se dibujan explícitamente. Obsérvese que los cuatro esquemas recursivos son idénticos realmente, con la única diferencia de que están girados 90°.

La forma básica de las curvas de Sierpinski es

$$S: A \rightarrow B \leftarrow C \leftarrow D \rightarrow \quad (3.21)$$

y los esquemas recursivos son

$$\begin{aligned} A: & A \rightsquigarrow B \Rightarrow D \rightarrow A \\ B: & B \leftarrow C \Downarrow A \rightsquigarrow B \\ C: & C \leftarrow D \Leftarrow B \leftarrow C \\ D: & D \rightarrow A \Updownarrow C \leftarrow D \end{aligned} \quad (3.22)$$

(Las flechas dobles designan líneas de longitud doble.)

Si se usan los mismos procedimientos básicos para las operaciones de dibujo que en el ejemplo de las curvas de Hilbert, el anterior esquema recursivo se transforma sin dificultad en un algoritmo (directa e indirectamente) recursivo.

```
procedure A(i: integer);
begin if i > 0 then
    begin A(i - 1); x := x + h; y := y - h; plot;
          B(i - 1); x := x + 2 * h; plot;
          D(i - 1); x := x + h; y := y + h; plot;
          A(i - 1)
    end
end
```

Este procedimiento se obtiene a partir de la primera línea del esquema recursivo (3.22). Pueden obtenerse en forma análoga procedimientos correspondientes a los esquemas B , C y D . El programa principal se construye de acuerdo con el esquema (3.21). Su misión es introducir los valores iniciales para las coordenadas del dibujo y determinar la longitud de la línea unitaria, h , acorde con el tamaño del papel, tal como se indica en el Programa 3.2. El resultado de ejecutar este programa para $n = 4$ aparece en la Fig. 3.7. Obsérvese que S_0 no se dibuja.

La elegancia obtenida con la utilización de la recursión en estos programas es obvia y convincente. La validez de los programas puede deducirse fácilmente de su estructura y esquemas de composición. Además, la utilización explícita

del parámetro de nivel i según el esquema (3.5) garantiza la terminación, dado que el nivel de recursión no puede llegar a ser superior a n . En contraste con esta formulación recursiva, los programas equivalentes que evitan la utilización explícita de la recursión son extremadamente farragosos y difíciles de entender.

Programa 3.2. Curvas de Sierpinski.

```

program Sierpinski (pf,output);
{dibujo de las curvas de Sierpinski de ordenes 1 a n}
const n = 4; h0 = 512;
var i,h,x,y,x0,y0: integer;
    fd: file of integer; {fichero del dibujo}
procedure A(i: integer);
begin if i > 0 then
    begin A(i-1); x := x+h; y := y-h; plot;
        B(i-1); x := x + 2*h; plot;
        D(i-1); x := x+h; y := y+h; plot;
        A(i-1)
    end
end ;
procedure B(i: integer);
begin if i > 0 then
    begin B(i-1); x := x-h; y := y-h; plot;
        C(i-1); y := y - 2*h; plot;
        A(i-1); x := x+h; y := y-h; plot;
        B(i-1)
    end
end ;
procedure C(i: integer);
begin if i > 0 then
    begin C(i-1); x := x-h; y := y+h; plot;
        D(i-1); x := x - 2*h; plot;
        B(i-1); x := x-h; y := y-h; plot;
        C(i-1)
    end
end ;
procedure D(i: integer);
begin if i > 0 then
    begin D(i-1); x := x+h; y := y+h; plot;
        A(i-1); y := y + 2*h; plot;
        C(i-1); x := x-h; y := y+h; plot;
        D(i-1)
    end
end ;

```

```

begin startplot
    i := 0; h := h0 div 4; x0 := 2*h; y0 := 3*h;
repeat i := i+1; x0 := x0-h;
    h := h div 2; y0 := y0+h;
    x := x0; y := y0; setplot;
    A(i); x := x+h; y := y-h; plot;
    B(i); x := x-h; y := y-h; plot;
    C(i); x := x-h; y := y+h; plot;
    D(i); x := x+h; y := y+h; plot;
until i = n;
endplot
end .

```

Programa 3.2. (Continuación)

Se insta al lector a que se convenza de esto, tratando de entender los programas que aparecen en la referencia [3-3].

3.4. ALGORITMOS DE «VUELTA ATRAS»

Un tema de programación particularmente curioso es la «resolución general de problemas». Su misión es definir algoritmos para encontrar soluciones de problemas específicos, sin seguir unas reglas prefijadas de cálculo, sino aplicando un método de tanteo sistemático («trial and error»). El procedimiento general es descomponer el proceso de tanteo en tareas parciales. Frecuentemente, estas tareas se expresan de la forma más natural recursivamente y están constituidas por la ejecución de un número finito de subtareas. Puede contemplarse el proceso general como un método de prueba o búsqueda que construye gradualmente e inspecciona y «poda» un árbol de tareas básicas. En muchos problemas este árbol de búsqueda crece muy rápidamente, habitualmente en forma exponencial, en función de un determinado parámetro. El esfuerzo de búsqueda crece en forma acorde. Normalmente el árbol de búsqueda puede reducirse mediante el uso exclusivo de técnicas heurísticas, disminuyendo con ello el esfuerzo computacional a límites tolerables.

No es misión de este texto estudiar las reglas heurísticas genéricas. Más bien se considera de interés en este capítulo el principio general de subdivisión de las tareas de resolución de problemas en subtareas y la aplicación de la recursión. Se empieza por la demostración de esta técnica mediante un ejemplo, en concreto, el ya bien conocido de *la vuelta del caballo*.

Se da un tablero de $n \times n$ con n^2 cuadros. Un caballo —que puede moverse según las reglas del ajedrez— se sitúa en el cuadro de coordenadas x_0, y_0 . Se pide encontrar, si existe, un recubrimiento del tablero completo, o sea, calcular un circuito de $n^2 - 1$ movimientos de forma que cada cuadro del tablero sea visitado exactamente una vez.

La forma obvia de reducir el problema de recubrimiento de n^2 cuadros es tomar como base el problema de, bien realizar un nuevo movimiento, o bien

decidir que ningún movimiento es posible. Por ello se define un algoritmo que trata de llevar a efecto un nuevo movimiento. Un primer intento aparece en (3.24).

```

procedure ensayar nuevo movimiento;
begin inicializar el conjunto de movimientos posibles;
repeat seleccionar el nuevo candidato de la lista de futuros movimientos;
  if aceptable then
    begin anotar movimiento;
      if tablero no lleno then
        begin ensayar nuevo movimiento;
          if no acertado then borrar la anotacion anterior
        end
      end
    until (movimiento acertado)  $\vee$  (no hay mas posibilidades)
end
(3.24)
```

Si se quiere ser más preciso al describir este algoritmo es obligatorio tomar decisiones sobre la forma de representar los datos. Un paso obvio es representar el tablero mediante una matriz, sea h su nombre. Se introduce también un tipo de datos para designar los valores de los índices:

```

type indice = 1 .. n;
var h: array [indice, indice] of integer
(3.25)
```

La decisión de representar cada cuadro del tablero por un valor entero en lugar de uno booleano, está motivada porque interesa conservar la historia de las ocupaciones sucesivas de cada cuadro. Es obvia la adopción del siguiente convenio:

```

h[x, y] = 0: el cuadro <x, y> no ha sido visitado
h[x, y] = i: el cuadro <x, y> ha sido visitado en el
               movimiento i-ésimo ( $1 \leq i \leq n^2$ )
(3.26)
```

La decisión siguiente a tomar corresponde a la elección de parámetros apropiados. Estos parámetros son necesarios para determinar las condiciones de arranque del movimiento siguiente a emprender y para reflejar si éste ha sido acertado. La primera tarea se resuelve adecuadamente especificando las coordenadas x, y , desde donde va a realizarse el movimiento, y especificando el número i del movimiento (con objeto de anotarlo). La segunda tarea requiere un parámetro booleano indicador de la calidad del resultado: $q = \text{true}$ significa acierto; $q = \text{false}$, fallo.

¿Qué instrucciones pueden ser detalladas ahora en base a estas decisiones? Ciertamente «tablero no lleno» puede expresarse mediante « $i < n^2$ ». Además, si se introducen dos variables locales u y v , que representen las coordenadas

de los destinos de posibles movimientos, determinadas de acuerdo con el tipo de salto del caballo, el predicado «aceptable» puede expresarse como la combinación lógica de las condiciones de que el nuevo cuadro esté en el tablero, o sea $1 \leq u \leq n$ y $1 \leq v \leq n$, y de que no haya sido ocupado previamente, es decir, $h[u, v] = 0$. La anotación del movimiento válido según estas reglas se expresa mediante la asignación $h[u, v] := i$, y el borrado de esta anotación mediante $h[u, v] := 0$. Si se introduce una variable local $q1$ y se utiliza como parámetro resultado en la llamada recursiva de este algoritmo, $q1$ puede sustituir a «movimiento acertado». De acuerdo con esto se llega a la formulación que aparece en (3.27)

```

procedure ensayar (i: integer; x, y: indice; var q: boolean);
var u, v: integer; q1: boolean;
begin inicializar el conjunto de movimientos posibles;
repeat sean u y v las coordenadas del siguiente movimiento, definido según
  las reglas del ajedrez;
  if ( $1 \leq u \leq n$ )  $\wedge$  ( $1 \leq v \leq n$ )  $\wedge$  ( $h[u, v] = 0$ ) then
    begin h[u, v] := i;
      if  $i < \text{sqr}(n)$  then
        begin ensayar( $i + 1$ , u, v, q1);
          if  $\neg q1$  then  $h[u, v] := 0$ 
        end else q1 := true
      end
    until q1  $\vee$  (no hay mas posibilidades);
    q := q1
  end
(3.27)
```

Basta un paso más para llegar a un programa expresado completamente en el lenguaje básico adoptado para la programación. Hay que hacer notar que hasta ahora el programa se ha desarrollado en forma totalmente independiente de las normas que rigen los saltos del caballo. Esta demora en la consideración de estas particularidades del problema ha sido totalmente deliberada. Pero ha llegado el momento de tenerlas en cuenta.

Dadas las coordenadas del punto de partida $\langle x, y \rangle$, hay ocho candidatos posibles para las coordenadas $\langle u, v \rangle$ del punto de destino. Se numeran de 1 a 8 en la Fig. 3.8.

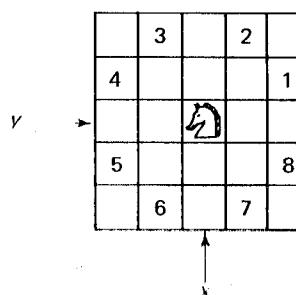


Fig. 3.8. Los ocho movimientos posibles del caballo.

Programa 3.3. Vuelta del caballo.

```

program vueltacaballo (output);
const n = 5; ncua = 25;
type indice = 1 .. n;
var i,j: indice;
q: boolean;
s: set of indice;
a,b: array [1 .. 8] of integer;
h: array [indice, indice] of integer;

procedure ensayar (i: integer; x,y: indice; var q: boolean);
  var k,u,v: integer; q1: boolean;
begin k := 0;
repeat k := k+1; q1 := false;
  u := x + a[k]; v := y + b[k];
  if (u in s)^(v in s) then
    if h[u,v] = 0 then
      begin h[u,v] := i;
      if i < ncua then
        begin ensayar (i+1,u,v,q1);
        if not q1 then h[u,v] := 0
        end else q1 := true
      end
    until q1 ∨ (k=8);
  q := q1
end {ensayar};

begin s := [1,2,3,4,5];
  a[1] := 2; b[1] := 1;
  a[2] := 1; b[2] := 2;
  a[3] := -1; b[3] := 2;
  a[4] := -2; b[4] := 1;
  a[5] := -2; b[5] := -1;
  a[6] := -1; b[6] := -2;
  a[7] := 1; b[7] := -2;
  a[8] := 2; b[8] := -1;
  for i := 1 to n do
    for j := 1 to n do h[i,j] := 0;
h[1,1] := 1; ensayar(2,1,1,q);
if q then
  for i := 1 to n do
    begin for j := 1 to n do write(h[i,j]:5);
      writeln
    end
  else writeln('NO HAY SOLUCION')
end .

```

Un método sencillo de obtener u, v a partir de x, y es por suma de las diferencias de coordenadas almacenadas bien en un array de pares de diferencias de coordenadas o en dos arrays de diferencias aisladas. Supóngase que estos arrays se denominan a y b y están convenientemente inicializados. Entonces, puede usarse un índice k para numerar el «nuevo candidato». Los detalles aparecen en el Programa 3.3. El procedimiento recursivo se inicia con una llamada que tiene como parámetro las coordenadas x_0, y_0 del cuadro a partir del cual comienza la vuelta del caballo. A este cuadro debe asignarse el valor 1; todos los demás pueden numerarse como vacíos.

$$h[x_0, y_0] := 1; \text{ ensayar}(2, x_0, y_0, q)$$

No hay que pasar por alto otro detalle. Una variable $h[u, v]$ existe, solamente, si tanto u como v están comprendidos dentro de los límites del array, $1 \dots n$. Consiguientemente la expresión de (3.27) que sustituye a «aceptable» en (3.24), sólo es válida si los dos primeros términos que la forman son ciertos. Una nueva formulación adecuada a estas condiciones aparece en el Programa 3.3, en el que, además, se ha sustituido la doble relación $1 \leq u \leq n$ por $u \in [1, 2, \dots, n]$, que, para n suficientemente pequeño, puede ser más eficiente (ver apartado 1.10.3). La Tabla 3.1 indica las soluciones obtenidas para posiciones iniciales $\langle 1, 1 \rangle$, $\langle 3, 3 \rangle$ para $n = 5$ y $\langle 1, 1 \rangle$ para $n = 6$.

Puede sustituirse el parámetro resultado q , y la variable local $q1$ por una variable global, simplificando con ello el programa, en cierta medida.

¿Qué abstracciones pueden hacerse basadas en este ejemplo? ¿Qué esquema exhibe que sea típico de esta clase de algoritmos de «resolución de problemas»? ¿Qué enseña? El rasgo característico es que se intentan pasos hacia la solución completa, que son anotados, y que tales anotaciones pueden borrarse más tarde al descubrir que el paso que dio origen a ellos puede no conducir a una solución completa, llevando en cambio a un «impasse». Esta acción se denomina «vuelta atrás». Su esquema general (3.28) viene de (3.24), suponiendo que el número potencial de candidatos en cada paso es finito.

En la realidad, los programas pueden, desde luego, adoptar varias formas, derivadas del esquema (3.28). Un esquema que aparece frecuentemente, utiliza un parámetro explícito de nivel, que indica el grado de recursión y permite definir una condición de terminación sencilla.

Sí, además, está fijado el número de candidatos a investigar en cada paso, por ejemplo, si es m , se puede aplicar el esquema derivado (3.29); se llama mediante la instrucción «ensayar (1)».

El resto de este capítulo se dedica al estudio de otros tres ejemplos. Cada uno pone de relieve distintas formas concretas del esquema abstracto (3.29) y se incluyen como nuevas ilustraciones del uso adecuado de la recursión.

1	6	15	10	21
14	9	20	5	16
19	2	7	22	11
8	13	24	17	4
25	18	3	12	23

23	10	15	4	25
16	5	24	9	14
11	22	1	18	3
6	17	20	13	8
21	12	7	2	19

1	16	7	26	11	14
34	25	12	15	6	27
17	2	33	8	13	10
32	35	24	21	28	5
23	18	3	30	9	20
36	31	22	19	4	29

Tabla 3.1. Tres vueltas de caballo.

```

procedure ensayar;
begin inicializar el conjunto de posibilidades;
repeat seleccionar el siguiente;
  if aceptable then
    begin anotarlo;
    if solución incompleta then
      begin ensayar el siguiente paso;
      if no acertado then cancelar la anotación
      end
    end
  until acertado ∨ no hay mas posibilidades
end

```

(3.28)

```

procedure ensayar (i: integer);
  var k: integer;
begin k := 0;
repeat k := k + 1; seleccionar la posibilidad k-esima;
  if aceptable then
    begin anotarla;
    if i < n then
      begin ensayar(i + 1);
      if no acertada then cancelar anotacion
      end
    end
  until acertado ∨ (k = m)
end

```

(3.29)

3.5. EL PROBLEMA DE LAS OCHO REINAS

El problema de las ocho reinas es un ejemplo bien conocido del uso de los métodos de tanteo sistemático («trial and error») y de los algoritmos de vuelta atrás. Fue investigado por C. F. Gauss en 1850 sin llegar a resolverlo completamente. Esto no es sorprendente, puesto que la propiedad característica de estos problemas es que no son adecuados para una solución analítica. En vez de ello, requieren grandes cantidades de trabajo paciente y preciso. Algoritmos de este tipo, por ello, han adquirido importancia, casi exclusivamente, gracias al computador, que posee estas cualidades en mayor grado que las personas, incluso los genios.

El problema de las ocho reinas se plantea de la forma siguiente (ver también la referencia [3-4]): Hay que situar ocho reinas en un tablero de ajedrez, de forma que ninguna reina pueda actuar sobre cualquiera de las otras.

Utilizando como patrón el esquema (3.29), se obtiene inmediatamente la siguiente versión general de una solución:

```

procedure ensayar(i: integer);
begin
  inicializar el conjunto de posiciones de la reina i-esima;
  repeat hacer la selección siguiente;
    if segura then
      begin ponerreina;
      if i < 8 then
        begin ensayar(i + 1);
        if no acertado then quitar reina
        end
      end
    until acertada ∨ no hay mas posiciones
end

```

(3.30)

Para continuar es preciso hacer algunas consideraciones sobre la representación de los datos. Dado que se sabe, por las reglas del ajedrez, que una reina actúa sobre todas las piezas situadas en la misma columna, fila o diagonal del tablero se deduce que cada columna puede contener una y sólo una reina, y que la elección de la situación de la reina i -ésima puede restringirse a los cuadros de la columna i . Por tanto, el parámetro i se convierte en el índice de columna, y por ello el proceso de selección de posiciones queda limitado a los ocho posibles valores del índice de fila j .

Queda pendiente la cuestión de cómo representar las ocho reinas en el tablero. Una elección obvia sería nuevamente una matriz cuadrada para representar el tablero. Sin embargo, una pequeña reflexión revela que este tipo de representación conduciría a operaciones bastante farragosas para comprobar la validez de las distintas posiciones. Esto no conviene en absoluto, ya que se trata de la operación ejecutada con más frecuencia. Debe elegirse, por tanto, una representación de datos que haga esta comprobación lo más simple que se pueda. Lo mejor es representar de la forma más directa posible aquella información que es realmente relevante y se utiliza con más frecuencia. En el caso que se estudia, esta información no es la posición de las reinas, sino el hecho de si una reina se ha situado ya o no a lo largo de cada fila y diagonal (se sabe ya que hay exactamente una situada en cada columna k siendo $1 \leq k \leq i$). Esto conduce a la siguiente elección de variables:

```
var x: array [1 .. 8] of integer;
   a: array [1 .. 8] of boolean;
   b: array [b1 .. b2] of boolean;
   c: array [c1 .. c2] of boolean; (3.31)
```

siendo

$x[i]$ la posición de la reina en la columna i ;
 $a[j]$ indicativo de que no hay reina en la fila j -ésima;
 $b[k]$ indicativo de que no hay reina en la diagonal $\swarrow k$ -ésima;
 $c[k]$ indicativo de que no hay reina en la diagonal $\searrow k$ -ésima.

La elección de los límites de los índices, $b1, b2, c1, c2$, viene obligada por la forma de cálculo de los índices de b y de c ; se observa que en una diagonal \swarrow todos los cuadros tienen idéntica suma de las coordenadas i, j , y que en una diagonal \searrow las diferencias de coordenadas $i - j$ son constantes. La solución adecuada aparece en el Programa 3.4.

A partir de estos datos, puede elaborarse la instrucción *ponerreina* en la forma:

```
x[i] := j; a[j] := false; b[i + j] := false; c[i - j] := false (3.32)
```

La instrucción *guitarreina* puede detallarse en la forma

```
a[j] := true; b[i + j] := true; c[i - j] := true (3.33)
```

y la condición *segura* se cumple si el cuadro $\langle i, j \rangle$ está situado en una fila y en unas diagonales todavía libres (lo que se representa por *true*). De aquí que pueda expresarse mediante la expresión lógica

```
a[j]  $\wedge$  b[i + j]  $\wedge$  c[i - j] (3.34)
```

Programa 3.4. Ocho reinas.

```
program ochorreinas1(output);
{encontrar una solución del problema de las ocho reinas}
var i: integer; q: boolean;
   a: array [ 1 .. 8] of boolean;
   b: array [ 2 .. 16] of boolean;
   c: array [-7 .. 7] of boolean;
   x: array [ 1 .. 8] of integer;

procedure ensayar(i: integer; var q: boolean);
  var j: integer;
begin j := 0;
repeat j := j + 1; q := false;
  if a[j]  $\wedge$  b[i + j]  $\wedge$  c[i - j] then
    begin x[i] := j;
      a[j] := false; b[i + j] := false; c[i - j] := false;
      if i < 8 then
        begin ensayar (i + 1, q);
          if  $\neg$  q then
            begin a[j] := true; b[i + j] := true; c[i - j] := true
            end
          end else q := true
        end
      end
  until q  $\vee$  (j = 8)
end {ensayar};

begin
  for i := 1 to 8 do a[i] := true;
  for i := 2 to 16 do b[i] := true;
  for i := -7 to 7 do c[i] := true;
  ensayar (1, q);
  if q then
    for i := 1 to 8 do write (x[i]: 4);
    writeln
end.
```

Con esto se completa el desarrollo de este algoritmo que aparece en su totalidad en el Programa 3.4. La solución calculada es $x = (1, 5, 8, 6, 3, 7, 2, 4)$ y aparece en la Fig. 3.9.

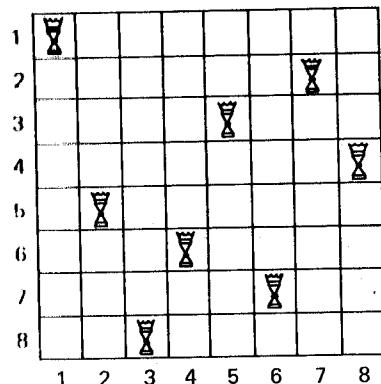


Fig. 3.9. Una solución del problema de las ocho reinas.

Antes de abandonar el tema del tablero de ajedrez, el ejemplo de las ocho reinas va a servir como ilustración de una extensión importante de un algoritmo de tanteo. La ampliación consiste —en términos generales— en encontrar no sólo una sino todas las soluciones del problema planteado.

Es sencillo incluir esta forma de ampliación. Hay que recordar que la generación de posibles candidatos debe progresar según un método sistemático que garantice que cada candidato aparezca sólo una vez. Esta propiedad del algoritmo se corresponde con una búsqueda sistemática en el árbol de candidatos, de tal forma que cada nodo sea visitado exactamente una vez. Permite —una vez encontrada una solución, y anotada debidamente— continuar simplemente con el siguiente candidato dado por el proceso sistemático de selección. El esquema general se basa en (3.29) y aparece en (3.35).

```
procedure ensayar(i: integer);
  var k: integer;
begin
  for k := 1 to m do
    begin seleccionar el candidato k-esimo; (3.35)
      if aceptable then
        begin anotarlo;
          if i < n then ensayar(i + 1) else imprimir solucion;
          cancelar anotacion
        end
      end
    end
end
```

Obsérvese que, debido a que la condición de terminación del proceso de selección se ha visto simplificada al término único $k = m$, la instrucción «repeat» ha sido acertadamente sustituida por una instrucción «for». Resulta sorprendente que la búsqueda de *todas* las soluciones posibles, se lleve a efecto mediante un programa más simple que el de búsqueda de una solución aislada.

El algoritmo extendido para determinar las 92 soluciones del problema de las ocho reinas se presenta en el Programa 3.5. Realmente, sólo hay 12 soluciones significativamente distintas, pero el programa no reconoce las simetrías. Las 12 soluciones generadas inicialmente se listan en la Tabla 3.2. Los números N que aparecen a la derecha indican la frecuencia de ejecución del test «cuadro seguro». Su valor medio en las 92 soluciones es 161.

Programa 3.5. Ocho reinas.

```
program ochorreinas (output);
var i: integer;
  a: array [ 1 .. 8 ] of boolean;
  b: array [ 2 .. 16 ] of boolean;
  c: array [-7 .. 7 ] of boolean;
  x: array [ -1 .. 8 ] of integer;

procedure print;
  var k: integer;
begin for k := 1 to 8 do write(x[k]: 4);
  writeln
end {print};

procedure ensayar(i: integer);
  var j: integer;
begin
  for j := 1 to 8 do
    if a[j] ^ b[i + j] ^ c[i - j] then
      begin x[i] := j;
        a[j] := false; b[i + j] := false; c[i - j] := false;
        if i < 8 then ensayar(i + 1) else print;
        a[j] := true; b[i + j] := true; c[i - j] := true
      end
    end {ensayar};
begin
  for i := 1 to 8 do a[i] := true;
  for i := 2 to 16 do b[i] := true;
  for i := -7 to 7 do c[i] := true;
  ensayar (1)
end .
```

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	N
1	5	8	6	3	7	2	4	876
1	6	8	3	7	4	2	5	264
1	7	4	6	8	2	5	3	200
1	7	5	8	2	4	6	3	136
2	4	6	8	3	1	7	5	504
2	5	7	1	3	8	6	4	400
2	5	7	4	1	8	6	3	72
2	6	1	7	4	8	3	5	280
2	6	8	3	1	4	7	5	240
2	7	3	6	8	5	1	4	264
2	7	5	8	1	4	6	3	160
2	8	6	1	3	5	7	4	336

Tabla 3.2. Doce soluciones del problema de las ocho reinas.

3.6. EL PROBLEMA DE LOS MATRIMONIOS ESTABLES

Se suponen dados dos conjuntos A y B , disjuntos, con igual número de elementos, n . Se pide encontrar un conjunto de n pares $\langle a, b \rangle$, tales que se verifique $a \in A$ y $b \in B$ y se satisfagan determinadas restricciones. Existen muchos criterios distintos para formar estos pares; uno de ellos es la regla denominada de «los matrimonios estables».

Supóngase que A es un conjunto de hombres y B es un conjunto de mujeres. Cada hombre y cada mujer han manifestado distintas preferencias por sus acompañantes. Si se eligen n parejas de forma que existan algún hombre y alguna mujer que, sin estar casados entre ellos, se prefieran mutuamente a sus parejas respectivas, se dice que la asignación es inestable. Si no existe ninguna pareja de este tipo se dice que es *estable*.

Esta situación caracteriza muchos problemas similares en los que deben realizarse asignaciones de acuerdo con determinadas preferencias tales como, por ejemplo, la elección de escuela por los estudiantes, la elección de los soldados por las diferentes armas del ejército, etc. El ejemplo de los matrimonios es particularmente intuitivo; obsérvese, sin embargo, que la lista de preferencias manifestada previamente permanece invariante y no cambia al hacer una determinada asignación. Esta regla simplifica el problema, pero también representa una distorsión de la realidad (conocida con el nombre de abstracción).

Una forma de buscar una solución es tratar de emparejar miembros de ambos conjuntos sucesivamente hasta que se agoten ambos. Si se desea obtener *todas* las asignaciones estables, puede esbozarse inmediatamente una solución utilizando el esquema de programa (3.35) como patrón. Sea $\text{ensayar}(h)$ el nombre del algoritmo para encontrar pareja para un hombre h y sea el orden de búsqueda el de la lista de preferencias manifestada por este hombre. La primera versión basada en estas hipótesis es (3.36)

```

procedure ensayar( $h$ : hombre);
  var  $r$ : rango;
begin
  for  $r := 1$  to  $n$  do
    begin tomar la preferencia  $r$ -ésima del hombre  $h$ ;
      if acceptable then
        begin anotar el matrimonio;
          if  $h$  no es el último hombre then ensayar(succ( $h$ ))
            else registrar el conjunto estable;
          cancelar el matrimonio
        end
      end
    end
end
(3.36)

```

Nuevamente se llega al punto en que no puede seguirse sin tomar previamente algunas decisiones sobre representación de datos. Se introducen tres tipos escalares y, por razones de simplificación, se toman como valores para ellos los enteros desde 1 a n . Aunque los tres tipos son formalmente idénticos, su distinta denominación contribuye significativamente a la claridad del programa. En particular, puede verse fácilmente para qué sirve cada variable.

```

type hombre = 1 .. n;
mujer = 1 .. n;
rango = 1 .. n
(3.37)

```

Los datos iniciales se representan por dos matrices que indican las preferencias de hombres y mujeres

```

var mhr: array [hombre, rango] of mujer
hmr: array [mujer, rango] of hombre
(3.38)

```

Por lo tanto, $mhr[h]$ designa la lista de preferencias del hombre h , es decir, $mhr[h][r] = mhr[h, r]$ es la mujer que ocupa el lugar r -ésimo en la lista de preferencias del hombre h . Análogamente, $hmr[m]$ es la lista de preferencias de la mujer m y $hmr[m, r]$ su r -ésima elección.

El resultado se representa por un array de mujeres x tal que $x[h]$ designa la pareja del hombre h . Para mantener la simetría —conocida también como «igualdad de derechos»— entre hombres y mujeres se introduce un array adicional y tal que $y[m]$ designa la pareja de la mujer m .

```

var x: array [hombre] of mujer;
y: array [mujer] of hombre
(3.39)

```

Evidentemente y no es estrictamente necesario dado que representa información

Rango	1	2	3	4	5	6	7	8	
Hombre 1 selecciona la mujer	7	2	6	5	1	3	8	4	
2	4	3	2	6	8	1	7	5	
3	3	2	4	1	8	5	7	6	
4	3	8	4	2	5	6	7	1	
5	8	3	4	5	6	1	7	2	
6	8	7	5	2	4	3	1	6	
7	2	4	6	3	1	7	5	8	
8	6	1	4	2	7	5	3	8	
Mujer	1 selecciona el hombre	4	6	2	5	8	1	3	7
2	8	5	3	1	6	7	4	2	
3	6	8	1	2	3	4	7	5	
4	3	2	4	7	6	8	5	1	
5	6	3	1	4	5	7	2	8	
6	2	1	3	8	7	4	6	5	
7	3	5	7	2	4	1	8	6	
8	7	2	8	4	5	6	3	1	

Tabla 3.3. Muestra tipo de datos de entrada para el problema de los matrimonios estables.

ya existente en x . De hecho, las relaciones

$$x[y[m]] = m, \quad y[x[h]] = h \quad (3.40)$$

se verifican para todos los h, m que están casados. Así pues, el valor $y[m]$ podría determinarse mediante una simple búsqueda en el array x ; sin embargo, el array y aumenta la eficiencia del algoritmo de una forma clara. La información representada por x e y se necesita para determinar la estabilidad de un conjunto de matrimonios dado. Dado que este conjunto se construye paso a paso casando individuos y comprobando la estabilidad después de cada matrimonio propuesto, x e y son necesarios incluso antes de que estén definidos todos sus componentes. Para llevar la cuenta de los componentes ya definidos pueden introducirse los arrays booleanos:

$$\begin{aligned} \text{soltero: array [hombre] of boolean} \\ \text{soltera: array [mujer] of boolean} \end{aligned} \quad (3.41)$$

con los significados siguientes:

- $\neg \text{soltero}[h]$ implica que $x[h]$ está definido,
- $\neg \text{soltera}[m]$ implica que $y[m]$ está definido.

Sin embargo, por simple inspección del algoritmo propuesto se ve que el estado civil de un hombre está determinado por el valor h de una forma sencilla:

$$\neg \text{soltero}[k] \equiv k < h \quad (3.42)$$

Esto indica que puede prescindirse del array $soltero$; consecuentemente se utiliza solamente el array $soltera$.

Las consideraciones anteriores conducen al proceso más elaborado (3.43). El predicado *aceptable* puede expresarse por la conjunción de *soltera* y *estable*, siendo esta última una función que requiere más elaboración.

```
procedure ensayar(h: hombre);
  var r: rango; m: mujer;
begin for r := 1 to n do
  begin m := mhr[h, r];
    if soltera[m]  $\wedge$  estable then
      begin x[h] := m; y[m] := h; soltera[m] := false;
        if h < n then ensayar(succ(h))
        else anotar conjunto estable;
        soltera[m] := true
      end
    end
  end
end
```

(3.43)

En este momento, todavía puede advertirse la gran semejanza de esta solución con el Programa 3.5.

La siguiente tarea fundamental es la elaboración más detallada del algoritmo que determina la estabilidad. Desgraciadamente, no es posible representar la estabilidad mediante una expresión tan simple como la empleada para la seguridad de la posición de una reina en el Programa 3.5. El primer detalle que ha de tenerse en cuenta es que la estabilidad es una consecuencia, por definición, de las comparaciones entre preferencias o rangos. Los rangos de hombres y mujeres no aparecen explícitamente en la colección de datos establecida hasta aquí. Desde luego, el rango de una mujer m en la mente de un hombre h puede calcularse, pero sólo mediante una costosa búsqueda de m en $mhr[h]$.

Dado que el cálculo de la estabilidad es una operación muy frecuente, es aconsejable hacer que esta información esté más directamente accesible. Con este objeto se introducen las dos matrices

$$\begin{aligned} rhm: \text{array [hombre, mujer] of rango;} \\ rmh: \text{array [mujer, hombre] of rango} \end{aligned} \quad (3.44)$$

tales que $rhm[h, m]$ designa el rango de m en la lista de preferencias del hombre h y $rmh[m, h]$ designa el rango del hombre h en la lista de m . Está claro que los valores de estos arrays auxiliares son constantes y pueden determinarse inicialmente a partir de los valores de mhr y hmr .

El proceso para determinar el predicado *estable* se define ahora estrictamente

de acuerdo con su definición original. Se recuerda que se está analizando la factibilidad del matrimonio entre h y m , siendo $m = mhr[h, r]$ es decir, m tiene rango r en la lista de preferencias de h . Optimistamente, se parte de la hipótesis de que lo normal es la estabilidad y se buscan posibles fuentes de perturbación en ella. ¿Dónde pueden estar? Hay dos posibilidades simétricas:

1. Puede haber una mujer mp , preferida a m por h , que a su vez prefiere h a su marido.
2. Puede haber un hombre hp , preferido a h por m que a su vez prefiere m a su mujer.

Para analizar la fuente de perturbaciones 1, se comparan los rangos $rmh[mp, h]$ y $rmh[mp, y[mp]]$ para todas las mujeres preferidas a m por h es decir, para todas las $mp = mhr[h, i]$, tales que $i < r$. Se sabe que todas estas mujeres posibles candidatas están ya casadas, ya que si alguna de ellas estuviera soltera, h la hubiera elegido ya. El proceso antes descrito puede formularse mediante una simple búsqueda lineal; e designa la estabilidad.

```
 $e := true; i := 1;$ 
 $\text{while } (i < r) \wedge e \text{ do}$ 
   $\begin{array}{l} \text{begin } mp := mhr[h, i]; i := i + 1; \\ \quad \text{if } \neg \text{soltera}[mp] \text{ then } e := rmh[mp, h] < rmh[mp, y[mp]] \end{array}$  (3.45)
   $\text{end}$ 
```

Para analizar la fuente de problemas 2, hay que investigar todos los candidatos hp preferidos por m a su actual asignación h , es decir, los hombres a investigar son todos los preferidos $hp = hmr[m, i]$, tales que $i < rmh[m, h]$. De forma análoga a la seguida para la fuente de problemas 1 es necesario comparar los rangos $rhm[hp, m]$ y $rhm[hp, x[hp]]$. Hay que tener cuidado de prescindir de las comparaciones que afecten a $x[hp]$ cuando hp esté todavía soltero. El control necesario para ello es una comprobación $hp < h$ dado que se sabe que todos los hombres anteriores a h están ya casados.

El algoritmo completo aparece en el Programa 3.6. La Tabla 3.3 es un conjunto de datos de entrada que representa los arrays mhr y hmr . Finalmente, la Tabla 3.4 da las nueve soluciones estables calculadas.

El algoritmo está basado notablemente en un esquema directo de vuelta atrás. Su eficiencia depende de la sofisticación del proceso de reducción («poda») del árbol. McVitie y Wilson [3-1 y 3-2] han presentado un algoritmo, en cierta forma más rápido, pero más complejo y menos claro, que han ampliado también al caso de conjuntos (de hombres y mujeres) de distinto tamaño.

Programa 3.6. Matrimonios estables.

```
program matrimonio (input, output);
{problema de los matrimonios estables}
const n = 8;
type hombre = 1 .. n; mujer = 1 .. n; rango = 1 .. n;
var h: hombre; m: mujer; r: rango;
  mhr: array [hombre, rango] of mujer;
  hmr: array [mujer, rango] of hombre;
  rhm: array [hombre, mujer] of rango;
  rmh: array [mujer, hombre] of rango;
  x: array [hombre] of mujer;
  y: array [mujer] of hombre;
  soltera: array [mujer] of boolean;

procedure imprimir;
  var h: hombre; rh, rm: integer;
begin rh := 0; rm := 0;
  for h := 1 to n do
    begin write (x[h]: 4);
      rh := rh + rhm[h, x[h]]; rm := rm + rmh[x[h], h]
    end;
    writeln (rh: 8, rm: 4);
end {imprimir};

procedure ensayar(h: hombre);
  var r: rango; m: mujer;

function estable: boolean;
  var hp: hombre; mp: mujer;
  i, lim: rango; e: boolean;
begin e := true; i := 1;
  while (i < r) \wedge e do
    begin mp := mhr[h, i]; i := i + 1;
      if \neg soltera[mp] then e := rmh[mp, h] < rmh[mp, y[mp]]
    end;
    i := 1; lim := rmh[m, h];
    while (i < lim) \wedge e do
      begin hp := hmr[m, i]; i := i + 1;
        if hp < h then e := rhm[hp, m] > rhm[hp, x[hp]]
      end;
      estable := e
    end {estable};
begin {ensayar}
  for r := 1 to n do
    begin m := mhr[h, r];
```

```

if soltera[m] then
  if estable then
    begin x[h] := m; y[m] := h; soltera[m] := false;
    if h < n then ensayar(succ(h)) else imprimir;
    soltera[m] := true
  end
end
end {ensayar};
begin {programa principal}
  for h := 1 to n do
    for r := 1 to n do
      begin read(mhr[h, r]); rhm[h, mhr[h, r]] := r
      end;
    for m := 1 to n do
      for r := 1 to n do
        begin read(hmr[m, r]); rmh[m, hmr[m, r]] := r
        end;
    for m := 1 to n do soltera[m] := true;
    ensayar (1)
end .

```

Programa 3.6. (Continuación)

Algoritmos del tipo de los dos últimos ejemplos, que generan todas las soluciones de un problema (dadas ciertas restricciones), se utilizan a menudo para seleccionar una o varias soluciones que son óptimas en algún sentido. En el ejemplo presente puede, por ejemplo, interesar aquella solución que en valor medio satisface más a los hombres —o a las mujeres— o a todas las personas.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	rh	rm	c^*
Solución	1	7	4	3	8	1	5	2	6	16	32
	2	2	4	3	8	1	5	7	6	22	27
	3	2	4	3	1	7	5	8	6	31	20
	4	6	4	3	8	1	5	7	2	26	22
	5	6	4	3	1	7	5	8	2	35	15
	6	6	3	4	8	1	5	7	2	29	20
	7	6	3	4	1	7	5	8	2	38	13
	8	3	6	4	8	1	5	7	2	34	18
	9	3	6	4	1	7	5	8	2	43	11
											758
											34

* c número de evaluaciones de estabilidad.

Solución 1 solución óptima masculina.

Solución 9 solución óptima femenina.

Tabla 3.4. Resultado del problema de los matrimonios estables.

Obsérvese que la Tabla 3.4 indica las sumas de los rangos de todas las mujeres en las listas de preferencias de sus maridos, y las sumas de los rangos de todos los hombres en las listas de preferencias de sus mujeres. Estos valores son

$$rh = \sum_{h=1}^n rhm[h, x[h]], \quad rm = \sum_{h=1}^n rmh[x[h], h] \quad (3.46)$$

La solución con mínimo valor rh se denomina solución masculina estable óptima; la de mínimo rm es la óptima estable femenina. Por el tipo de estrategia de búsqueda elegido se generan primeramente las soluciones buenas desde el punto de vista masculino y, al final del proceso, las soluciones buenas desde una perspectiva femenina. En este sentido, el algoritmo está desplazado del lado de la población masculina. Esto puede modificarse rápidamente intercambiando sistemáticamente los papeles de hombres y mujeres, es decir, intercambiando simplemente hmr y mhr , y rh y rm .

No se considera de interés ampliar más este programa, y se deja la incorporación de la búsqueda de una solución óptima para el próximo y último ejemplo de un algoritmo de vuelta atrás

3.7. EL PROBLEMA DE LA SELECCION OPTIMA

El último ejemplo de algoritmo de vuelta atrás es una extensión lógica de los dos ejemplos previos representados por el esquema (3.35). En primer lugar se utilizó el principio de vuelta atrás para encontrar una solución *aislada* de un problema dado. Los problemas de la vuelta del caballo y de las ocho reinas fueron ejemplos de esto. A continuación se planteó el objetivo de encontrar *todas* las soluciones de un problema dado. Los problemas de las ocho reinas y los matrimonios estables fueron ejemplos de este nuevo enfoque. Ahora se trata de encontrar la *solución óptima*.

A este efecto, es necesario generar todas las soluciones posibles y, en el proceso de generación, retener aquella que es óptima en algún sentido. Suponiendo que el óptimo se define en base a una función $f(s)$ positiva, el algoritmo se obtiene a partir del esquema 3.35 sustituyendo la instrucción *imprimir solución* por

if $f(solución) > f(optimo)$ then $optimo := solución$ (3.47)

La variable *optimo* almacena la mejor solución encontrada hasta el momento. Naturalmente, hay que inicializarla adecuadamente. Además, es corriente almacenar también el valor $f(optimo)$ en otra variable para evitar repetir su cálculo, lo que sería muy frecuente en el proceso.

A continuación se presenta un ejemplo del problema general de encontrar una solución óptima de un problema dado: Se escoge el importante y muy habitual problema de hacer una *selección óptima* entre un conjunto de objetos, teniendo en cuenta determinadas restricciones. Se construyen gradualmente selecciones que constituyen soluciones aceptables, investigando objetos indivi-

duales del conjunto de base. Un procedimiento *ensayar* describe el proceso de investigar si es adecuado incluir un determinado objeto, y este se llama recursivamente (para investigar el objeto siguiente) hasta tener en cuenta todos los objetos del conjunto.

Se observa que la consideración de cada objeto (denominado candidato en los ejemplos precedentes) tiene dos resultados posibles: bien la *inclusión* del objeto investigado en la selección que está en curso, o bien su *exclusión*. Esto hace que no sea adecuado el uso de las instrucciones **for** o **repeat**; en su lugar debe indicarse cada caso explícitamente. Esto se presenta en (3.48) (se supone que los objetos están numerados 1, 2, ..., n).

```

procedure ensayar(i: integer);
begin
 1: if inclusion es aceptable then
    begin incluir el objeto i-esimo;
      if i < n then ensayar(i + 1) else ver si es optimo;
      eliminar el objeto i-esimo
    end;
 2: if exclusion es aceptable then
    if i < n then ensayar(i + 1) else probar optimo
end

```

(3.48)

A partir de este esquema es evidente que hay 2^n conjuntos posibles; está claro que deben emplearse criterios apropiados de aceptabilidad para reducir muy drásticamente el número de candidatos investigados. Para esclarecer este proceso se elige un ejemplo concreto de problema de selección: Supóngase que cada uno de los n objetos a_1, \dots, a_n está caracterizado por su peso p y su valor v . Supóngase que el conjunto óptimo es aquél en que la suma de valores de sus componentes es máxima, y que como restricción se introduce un límite a la suma de sus pesos. Este es un problema muy conocido por los viajeros que hacen las maletas seleccionando entre n artículos de forma que su valor total sea óptimo y su peso no exceda de un valor permitido.

En base a las consideraciones anteriores, es posible decidir ahora la representación, en forma de datos, de los hechos dados. Las especificaciones (3.49) se deducen fácilmente de las consideraciones anteriores.

```

type indice = 1 .. n;
objeto = record p, v: integer end
var a: array [indice] of objeto;
limp, totv, maxv: integer;
s, opts: set of indice

```

(3.49)

Las variables *limp* y *totv* designan el límite de peso y el valor total de los n objetos. Estos dos valores son en realidad constantes durante todo el proceso de selección, y representan la selección de objetos que se está procesando en la que cada

objeto se representa por su nombre (índice). *opts* es la selección óptima encontrada hasta el momento y *maxv* su valor correspondiente.

¿Cuáles son los criterios a emplear para considerar que un objeto es aceptable en la selección que está en curso de proceso? Si se considera la *inclusión* de un objeto, éste es seleccionable si entra dentro del peso permitido. Si no entra, se puede finalizar el intento de añadir nuevos objetos a la selección en curso. Si se considera la *exclusión* de un objeto, el criterio de aceptabilidad, es decir, el criterio para continuar la construcción de la selección en curso, es que el valor total que sea todavía alcanzable después de esta exclusión no sea menor que el óptimo encontrado hasta el momento. Ya que, si es menor, la continuación de la búsqueda, aunque pueda producir algunas soluciones, no producirá la óptima. Por lo tanto, cualquier búsqueda adicional por el camino en curso será infructuosa. A partir de estas dos condiciones se determinan las cantidades relevantes a calcular en cada etapa del proceso de selección:

1. El peso total *pt* alcanzado por la selección *s* obtenida hasta el momento.
2. El valor, *va*, todavía alcanzable por la selección en curso *s*.

Ambas entidades se representan adecuadamente como parámetros del procedimiento *ensayar*.

La condición *inclusion es aceptable* de (3.48) puede formularse teniendo en cuenta lo anterior en la forma

$$pt + a[i] \cdot p \leq limp \quad (3.50)$$

y la comprobación subsiguiente de óptimo, en la forma

```

if va > maxv then
begin {nuevo optimo, anotarlo}
  opts := s; maxv := va
end

```

(3.51)

La última asignación se basa en que el valor alcanzable *coincide* con el valor alcanzado, una vez se hayan tratado los n objetos.

La condición *exclusion es aceptable* de (3.48) se expresa en la forma

$$va - a[i] \cdot v > maxv \quad (3.52)$$

Dado que después se utiliza nuevamente el valor $va - a[i] \cdot v$, se le da el nombre *val* con objeto de evitar repetir su cálculo.

A continuación se presenta el programa completo, construido a partir de los bloques que van desde (3.48) hasta (3.52), a los que se añaden las instrucciones adecuadas de inicialización de las variables globales. La facilidad con que se expresan la inclusión y exclusión del conjunto *s* mediante el uso de operadores

de conjuntos es notable. Los resultados de la ejecución del Programa 3.7 con pesos permitidos variando entre 10 y 120 se listan en la Tabla 3.5.

Este esquema de algoritmo de marcha atrás con un factor de limitación del crecimiento potencial del árbol de búsqueda, se conoce también como *algoritmo «Branch and Bound»*.

Programa 3.7. Selección óptima.

```

program seleccion (input, output);
{encontrar una selección óptima de objetos con una restricción}
const n = 10;
type indice = 1 .. n;
objeto = record v, p: integer end;
var i: indice;
a: array [indice] of objeto;
limp, totv, maxv: integer;
p1, p2, p3: integer;
s, opts: set of indice;
z: array [boolean] of char;
procedure ensayar(i: indice; pt, va: integer);
var val: integer;
begin {ensayar la inclusión del objeto i}
if pt + a[i].p ≤ limp then
begin s := s + [i];
if i < n then ensayar(i + 1, pt + a[i].p, va) else
if va > maxv then
begin maxv := va; opts := s
end;
s := s - [i]
end;
{ensayar la exclusión del objeto i} val := va - a[i].v;
if val > maxv then
begin if i < n then ensayar(i + 1, pt, val) else
begin maxv := val; opts := s
end
end
end {ensayar};
begin totv := 0;
for i := 1 to n do
with a[i] do
begin read(p, v); totv := totv + v
end;
read(p1, p2, p3);
z[true] := '>'; z[false] := '<';

```

```

write(' PESO ');
for i := 1 to n do write(a[i].p: 4);
writeln; write(' VALOR ');
for i := 1 to n do write(a[i].v: 4);
writeln;
repeat limp := p1; maxv := 0; s := []; opts := [];
ensayar(1, 0, totv);
write(limp);
for i := 1 to n do write(' ', z[i in opts]);
writeln; p1 := p1 + p2
until p1 > p3
end.

```

Programa 3.7. (Continuación)

Peso	10	11	12	13	14	15	16	17	18	19
Valor	18	20	17	19	25	21	27	23	25	24
10	*									
20								*		
30					*			*		
40	*				*			*		
50	*	*		*				*		
60	*	*	*	*	*					
70	*	*			*			*		*
80	*	*	*		*		*	*	*	
90	*	*			*		*		*	*
100	*	*		*	*		*	*	*	
110	*	*	*	*	*	*	*		*	
120	*	*			*	*	*	*	*	*

Tabla 3.5. Resultado típico del programa de selección óptima.

E J E R C I C I O S

- 3.1. (Problema de las torres de Hanoi). Se dan tres barras verticales y n discos de diferentes tamaños. Los discos pueden apilarse sobre las barras formando «torres». Se suponen los n discos colocados inicialmente en la barra A en orden de tamaño decreciente, tal como aparece en la Fig. 3.10 para $n = 3$. La tarea a realizar es mover los n discos desde la barra A hasta la barra C de forma que queden ordenados de la misma forma inicial. Esto debe realizarse bajo las restricciones:

1. En cada paso se mueve exactamente un disco desde una barra a otra.
2. En ningún momento puede situarse un disco encima de otro más pequeño.
3. Puede utilizarse la barra B como almacén auxiliar.

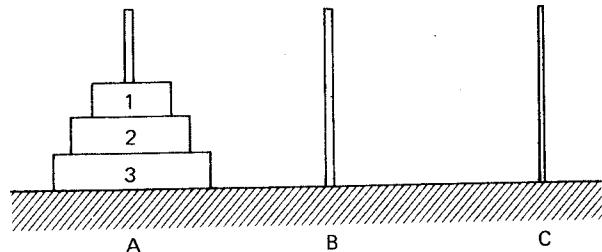


Fig. 3.10. Torres de Hanoi.

Se pide encontrar un algoritmo que lleve a efecto esta tarea. Obsérvese que una torre puede considerarse apropiadamente como formada por el disco único situado en la base y la torre constituida por los discos restantes. Describir el algoritmo mediante un programa recursivo.

- 3.2. Escribir un procedimiento que genere las $n!$ permutaciones de n elementos a_1, \dots, a_n sobre el mismo área de memoria sin recurrir a otro array. Al generar la permutación siguiente a una dada, hay que llamar un procedimiento paramétrico Q que, por ejemplo, puede imprimir la permutación generada.

Nota: Considérese la tarea de generar todas las permutaciones de los elementos a_1, \dots, a_m como formada por m sub-tareas de generación de todas las permutaciones de a_1, \dots, a_{m-1} a las que se añade a_m , habiendo intercambiado inicialmente en la subtarea i -ésima los elementos a_i y a_m .

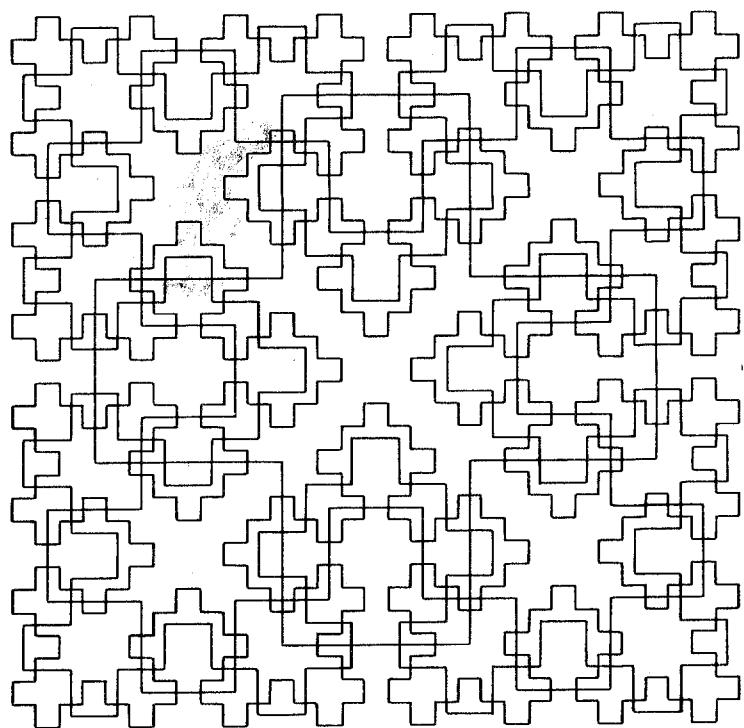
- 3.3. Deducir el esquema recursivo de la Fig. 3.11 que es una superposición de las cuatro curvas W_1, W_2, W_3, W_4 . La estructura es semejante a la de las curvas de Sierpinski (3.21) y (3.22). A partir del esquema recursivo, se pide obtener un programa recursivo que dibuje estas curvas.

- 3.4. Sólo 12 de las 92 soluciones calculadas por el Programa 3.5 de las ocho reinas, son esencialmente diferentes. Las otras pueden obtenerse mediante simetrías respecto de ejes o respecto del centro. Se pide diseñar un programa que determine las 12 soluciones principales. Obsérvese que, por ejemplo, la búsqueda en la columna 1 puede limitarse a las posiciones 1-4.

- 3.5. Cambiar el programa de los matrimonios estables de forma que determine la solución óptima (masculina o femenina). Debe convertirse por ello en un programa «branch and bound» del tipo representado por el Programa 3.7.

- 3.6. Una determinada compañía de ferrocarril sirve n estaciones E_1, \dots, E_n . Esta compañía quiere mejorar el servicio de información a clientes mediante terminales de información por computador. Un cliente escribe en un terminal su estación de salida E_A y su estación de destino E_B y se considera que va a recibir (inmediatamente) el programa de conexiones por tren con tiempo total de viaje mínimo.

Diseñar un programa para calcular la información deseada. Supóngase que el horario (que es el banco de datos que sirve de base) está disponible en una estructura de datos adecuada que contiene las horas de salida (llegada) de todos los trenes disponibles. Naturalmente, no todas las estaciones están conectadas por líneas directas (ver también el Ejercicio 1.8).

Fig. 3.11. Curvas W de órdenes 1 a 4.

- 3.7. La función del Ackerman A está definida para todos los valores enteros no negativos m y n de la forma siguiente:

$$A(0, n) = n + 1$$

$$A(m, 0) = A(m - 1, 1) \quad (m > 0)$$

$$A(m, n) = A(m - 1, A(m, n - 1)) \quad (m, n > 0)$$

Se pide diseñar un programa que calcule $A(m, n)$ sin utilizar la recursión.

Utilizar como guía el Programa 2.11, versión no recursiva del método rápido de ordenación. Diseñar un conjunto de reglas para transformar los programas recursivos en programas iterativos.

REFRENCIAS

- 3.1. McVITIE, D. G. y WILSON, L. B., «The Stable Marriage Problem», *Comm. ACM*, **14**, No. 7 (1971), 486-92.
- 3.2., «Stable Marriage Assignment for Unequal Sets», *BIT*, **10**, (1970), 295-309.

- 3-3. «Space Filling Curves, or How to Waste Time on a Plotter», *Software-Practice and Experience*, 1, No. 4 (1971), 403-40.
- 3-4. WIRTH, N., «Program Development by Stepwise Refinement», *Comm. ACM*, 14, No. 4 (1971), 221-27.

4

ESTRUCTURAS DINAMICAS DE INFORMACION

4.1. TIPOS RECURSIVOS DE DATOS

Como estructuras fundamentales de datos, en el Cap. 2 se introdujeron las estructuras array, registro, y conjunto. Se les llama fundamentales porque constituyen los elementos con los cuales se pueden formar estructuras más complejas, y porque en la práctica son las que con más frecuencia se presentan. El interés de definir un tipo de datos, y especificar a continuación que ciertas variables son de tal tipo, estriba en el hecho de que el campo de valores que pueden tomar estas variables, y por lo tanto su representación en memoria, quedan fijados para siempre. Es por ello que variables declaradas de esta forma son llamadas *estáticas*. Sin embargo, hay muchos problemas que requieren estructuras de información bastante más complejas. La característica de estos problemas es que sus estructuras cambian durante la computación. Estas se llaman, por tanto, estructuras *dinámicas*. Naturalmente, los componentes de tales estructuras son —a cierto nivel de detalle— *estáticos*, es decir, de uno de los tipos fundamentales de datos. Este capítulo se dedica a la construcción, análisis, y gestión de estructuras dinámicas de información.

Es importante señalar que existen grandes analogías entre los métodos utilizados para estructurar algoritmos y los utilizados para estructurar datos. Como en todas las analogías, existen también algunas diferencias (si no, serían identidades), pero de todas formas una comparación de los métodos de estructurar programas y datos resulta esclarecedora.

La instrucción elemental, no estructurada, es la asignación. Su correspondiente en la familia de estructuras de datos es el tipo escalar, no estructurado. Estos son los dos elementos atómicos con los cuales se pueden construir instrucciones genéricas y tipos de datos. Las estructuras más simples, obtenidas mediante enumeración o sucesión, son la instrucción compuesta y la estructura registro.

Ambas estructuras están formadas por un número finito (normalmente pequeño) de componentes, enumerados explícitamente, los cuales pueden ser todos diferentes unos de otros. Si todos los componentes son idénticos, no necesitan ser especificados individualmente: se utilizan la instrucción **for** y la estructura **array** para indicar multiplicación por un factor finito conocido. Una elección entre dos o más variantes se expresa por la instrucción condicional o la instrucción **case** y por la estructura registro con variante, respectivamente. Y, por último, una repetición por un factor inicialmente desconocido (y potencialmente infinito) se expresa por medio de las instrucciones **while** o **repeat**. La estructura de datos correspondiente es la secuencia (fichero), que es la estructura más simple que permite la construcción de tipos de cardinalidad infinita.

Surge la pregunta de si existe o no una estructura de datos que se corresponda con la instrucción procedimiento. Naturalmente, la propiedad más interesante y original de los procedimientos a este respecto es la *recursión*. Valores de un tipo de datos recursivo contendrían uno o más componentes pertenecientes a su mismo tipo, de una forma análoga a como un procedimiento contiene una o más llamadas a sí mismo. De la misma manera que los procedimientos, tales definiciones de tipos de datos podrían ser directa o indirectamente recursivas.

Un ejemplo sencillo de un objeto que sería representado muy apropiadamente como un tipo definido recursivamente es la expresión aritmética de los lenguajes de programación. La recursión se utiliza para reflejar la posibilidad de jerarquizar expresiones, es decir, de utilizar subexpresiones entre paréntesis como operandos de expresiones. Por lo tanto, se define informalmente una expresión como:

Una *expresión* está formada por un término, seguido de un operador, seguido de un término. (Los dos términos constituyen los operandos del operador.) Un *término* es, bien una variable —representada por un identificador—, o una expresión que está entre paréntesis.

Se puede describir fácilmente un tipo de datos cuyos valores representan tales expresiones, utilizando las herramientas ya disponibles, con la adición de la *recursión*:

```
type expresion = record op: operador;
    opd1, opd2: termino
  end;
type termino = record
  if t then (id: alfa)
  else (subex: expresion)
end
```

 (4.1)

Se observa que cada variable del tipo *termino* está formada por dos componentes, el indicador *t* y, si *t* es cierto, el campo *id*, si no, el campo *subex*. Consideren-

se ahora, por ejemplo, las siguientes cuatro expresiones:

1. $x + y$
 2. $x - (y * z)$
 3. $(x + y) * (z - w)$
 4. $(x/(y + z)) * w$
- (4.2)

Estas expresiones pueden representarse por los dibujos de la Fig. 4.1, que muestran su estructura recursiva, y determinan la representación de estas expresiones en memoria.

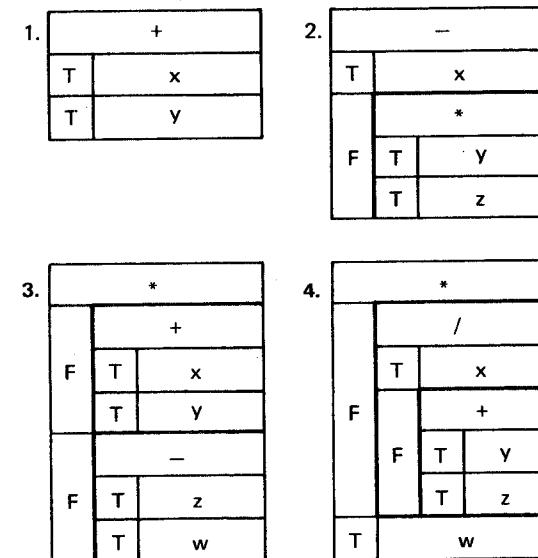


Fig. 4.1. Imagen en memoria de estructuras recursivas registro.

Un segundo ejemplo de estructura de información recursiva es el árbol genealógico: se define un *árbol* por (el nombre de) una persona y los dos árboles de sus padres. Esta definición conduce inevitablemente a una estructura infinita. Los árboles genealógicos reales son finitos porque a cierto nivel los antepasados se desconocen. Este hecho se puede tener en cuenta utilizando una estructura con variante tal como se muestra en (4.3).

```
type arb = record
  if conocido then
    (nombre: alfa;
    padre, madre: arb)
  end
```

 (4.3)

(Obsérvese que cada variable del tipo *arb* tiene al menos un componente, que es el indicador llamado *conocido*. Si su valor es cierto, entonces hay tres campos más; si no, ninguno más.) El constructor (reursivo) registro

$$x = (T, Ted, (T, Fred, (T, Adam, (F), (F)), (F)), (F), (F)) \\ (T, Mary, (F), (T, Eva, (F), (F)))$$

designa el valor que se muestra en la Fig. 4.2 de tal forma que sugiere una posible representación en memoria. (Puesto que sólo se trata de un único tipo de registro, se ha omitido el identificador del tipo *arb* precediendo cada constructor.)

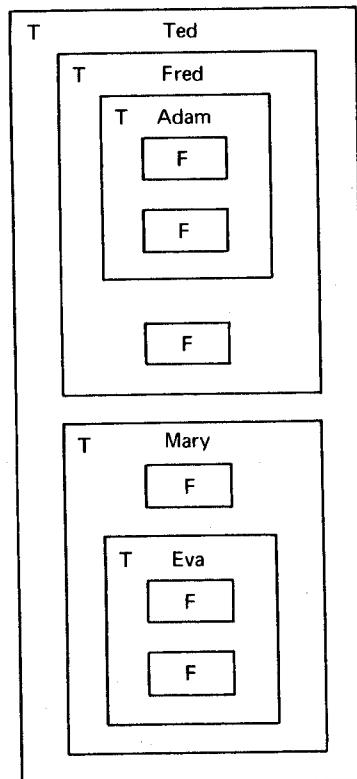


Fig. 4.2. Estructura árbol genealógico.

La importancia que tiene el campo variante se manifiesta de una forma clara; es el único mecanismo mediante el cual se puede acotar una estructura recursiva de datos, y es por lo tanto un compañero inevitable de toda definición recursiva. La analogía entre conceptos de estructuración de datos y de estructuración de programas es especialmente notable en este caso. Todo procedimiento recursivo debe contener necesariamente una instrucción condicional para que su ejecución pueda terminar. La terminación del proceso se corresponde evidentemente con la cardinalidad finita.

4.2. PUNTEROS O REFERENCIAS

La facultad de variar su tamaño es la propiedad característica de las estructuras recursivas que claramente les distingue de las estructuras fundamentales (arrays, registros, conjuntos). Por tanto, no es posible asignar una cantidad fija de memoria para una estructura definida recursivamente y, como consecuencia, un compilador no puede asociar direcciones explícitas con los componentes de tales variables. La técnica que se utiliza más frecuentemente para resolver este problema consiste en realizar una *asignación dinámica* de memoria, es decir, asignación de memoria para los componentes individuales al tiempo que éstos son creados durante la ejecución del programa, en vez de en el momento de la compilación del mismo. El compilador asigna una cantidad fija de memoria para mantener la dirección del componente asignado dinámicamente en vez del componente en sí. Por ejemplo, el árbol genealógico mostrado en la Fig. 4.2 estaría representado por registros individuales, muy posiblemente no adyacentes, cada uno de ellos representando una persona. Estas personas son entonces enlazadas asignando sus direcciones a los campos «padre» y «madre» respectivos. Gráficamente, la mejor forma para expresar esta situación es mediante el uso de flechas o punteros (ver Fig. 4.3).

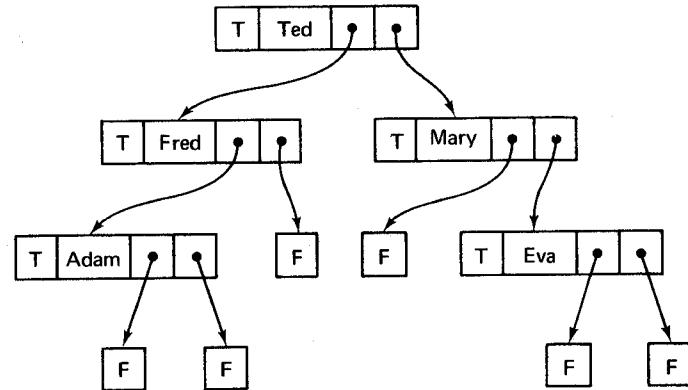


Fig. 4.3. Estructura enlazada con punteros.

Hay que resaltar que el uso de punteros para realizar estructuras recursivas es meramente una técnica. El programador no necesita ser consciente de su existencia. La asignación de memoria puede realizarse automáticamente la primera vez que un componente nuevo es referenciado. Sin embargo, si la técnica de utilizar referencias o punteros se hace explícita, se pueden construir estructuras de datos más generales que aquellas definibles por simples mecanismos recursivos. En particular, es posible definir estructuras circulares o «infinitas» y obligar a que ciertas estructuras estén *compartidas*. Esta es la razón de que en los lenguajes

de programación avanzados se permite la manipulación explícita de las referencias a los datos además de los datos en sí. Esto implica que debe haber una clara distinción notacional entre datos y referencias a datos y que, consecuentemente, se deben introducir tipos de datos cuyos valores sean punteros (referencias) a otros datos. La notación que se va a utilizar a este fin es la siguiente:

$$\text{type } T_p = \uparrow T \quad (4.4)$$

La declaración de tipo (4.4) expresa que los valores del tipo T_p son punteros a datos de tipo T . Por lo tanto, la flecha de (4.4) se lee como «puntero a». Es muy importante que el tipo de elementos a los que se apunta se haga patente en la declaración de T_p . Se dice que T_p está *ligado* a T . Esta ligadura distingue a los punteros de los lenguajes de alto nivel de las direcciones en código ensamblador, y es un mecanismo de importancia fundamental para mejorar la seguridad en la programación utilizando redundancia en la notación empleada.

Los valores de tipo puntero se generan cuando a un dato se le asigna memoria dinámicamente. Se adoptará el convenio de que tales ocasiones sean mencionadas explícitamente siempre que ocurran, en vez de suponer que un elemento resulta automáticamente asignado la primera vez que se le menciona en el programa. A tal fin se introduce el procedimiento intrínseco *new*. Si p es una variable puntero del tipo T_p , la instrucción

$$\text{new}(p) \quad (4.5)$$

crea (asigna memoria a) una variable del tipo T , genera un puntero del tipo T_p que referencia esta nueva variable, y asigna este puntero a la variable p (ver Figura 4.4). El valor puntero puede ahora ser referenciado como p (es decir, como el valor de la variable puntero p). Por otra parte, la variable referenciada por p se denomina $p\uparrow$. Esta es la variable del tipo T creada dinámicamente.

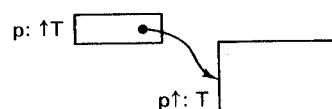


Fig. 4.4. Creación dinámica de la variable p .

Ya se ha mencionado que es esencial que exista un componente variante en todo tipo recursivo, para asegurar cardinalidad finita. El ejemplo de árbol genealógico tiene una estructura cuyo tipo se presenta muy frecuentemente [ver (4.3)], es decir, el caso en el cual el campo discriminante puede tomar dos valores (booleano) y, cuando el valor resulta ser *false*, esto indica que no existen más componentes. Esto se puede expresar mediante el *esquema de declaraciones* (4.6).

$$\text{type } T = \text{record if } p \text{ then } S(T) \text{ end} \quad (4.6)$$

$S(T)$ designa una secuencia de definiciones de campos que incluye uno o más campos de tipo T , con los cuales se asegura la recursividad. Todas las estructuras del tipo (4.6) tendrán una forma de árbol (o lista) similar a la mostrada en la Figura 4.3. Es peculiar de estas estructuras contener punteros a componentes de datos que sólo tienen un campo discriminador, es decir, sin campos de información. El uso de punteros como técnica para realizar estructuras recursivas sugiere una forma sencilla de ahorrar memoria dejando que la información del campo discriminador esté incluida en el valor del puntero mismo.

La solución comúnmente adoptada consiste en *ampliar* el campo de valores de un tipo T_p con un valor adicional que no apunta a ningún elemento. Se designa este valor por el símbolo especial *nil*, y se entiende que *nil* es automáticamente un elemento de todos los tipos puntero declarados. Esta extensión del campo de los valores punteros explica cómo se pueden generar estructuras finitas *sin* la presencia explícita de variantes (condiciones) en su declaración (recursiva).

En (4.7) y (4.8), respectivamente, se dan las nuevas declaraciones de los tipos de datos de (4.1) y (4.3) con punteros explícitos. Nótese que en el caso (4.8), que originalmente corresponde al esquema (4.6), el componente variante del registro ha desaparecido, puesto que $p\uparrow \cdot \text{conocido} = \text{false}$ se expresa ahora como $p = \text{nil}$. El cambio de nombre de *arbol* a *persona* refleja la diferencia de significado que conlleva la introducción de valores puntero explícitos. En vez de considerar primero la estructura en toda su amplitud, e investigar a continuación su subestructura y sus componentes, ahora se atiende a sus componentes en primer lugar y su interrelación (representada por punteros) no se manifiesta de una forma directa.

$$\begin{aligned} \text{type } \text{expresion} &= \text{record op: operador;} \\ &\quad \text{opd1, opd2: } \uparrow \text{termino} \\ &\text{end;} \end{aligned} \quad (4.7)$$

$$\begin{aligned} \text{type } \text{termino} &= \text{record} \\ &\quad \text{if } t \text{ then (id: alfa)} \\ &\quad \text{else (sub: } \uparrow \text{expresion)} \\ &\text{end} \end{aligned} \quad (4.8)$$

$$\begin{aligned} \text{type } \text{persona} &= \text{record nombre: alfa;} \\ &\quad \text{padre, madre: } \uparrow \text{persona} \\ &\text{end} \end{aligned} \quad (4.8)$$

En la Fig. 4.5 se representa la estructura del árbol genealógico mostrado en las Figs. 4.2 y 4.3, donde los punteros a personas desconocidas se denotan por *nil*. Puede verse claramente cuál es el ahorro de memoria obtenido.

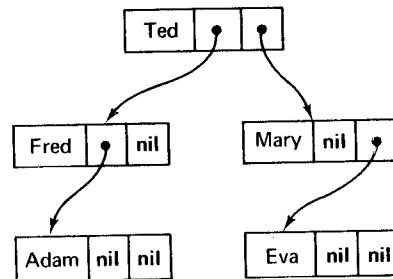


Fig. 4.5. Estructura con punteros nil.

En referencia a la misma Fig. 4.5, supóngase que Fred y Mary son hermanos. Esta situación puede expresarse fácilmente cambiando los dos valores **nil** en los campos respectivos de los dos registros. Una implementación que ocultase el concepto de punteros o utilizase una técnica distinta para manejo de la memoria forzaría al programador a representar los registros de Adam y Eva por duplicado. Aunque a la hora de acceder a los datos para inspección no importa si los dos padres (y las dos madres) están duplicados o representados por un único registro, la diferencia es *esencial a la hora de actualizar selectivamente*. Tratando los punteros como datos explícitos en vez de como detalles ocultos de la representación permite que el programador exprese claramente cuando desea *compartir memoria*.

Una consecuencia adicional del uso explícito de punteros es la posibilidad de definir y manipular estructuras cíclicas de datos. Esta posibilidad adicional no sólo aumenta el poder expresivo del programador, sino que también requiere mayor cuidado en su trabajo puesto que la manipulación de estructuras cíclicas de datos puede fácilmente conducir a procesos que no acaben nunca.

Este fenómeno de relación íntima entre flexibilidad y poder expresivo, y el peligro de utilización incorrecta de los mismos, es bien conocido en programación y recuerda especialmente la instrucción **goto**. De hecho, si se quiere extender la analogía entre estructuras de programas y estructuras de datos, la estructura recursiva pura de datos podría bien colocarse al nivel correspondiente al procedimiento, mientras que la introducción de punteros es comparable al uso de instrucciones **goto**. De la misma forma que la instrucción **goto** permite la construcción de cualquier tipo de programa (incluyendo bucles), los punteros permiten la construcción de cualquier tipo de estructura de datos (incluyendo ciclos). En la Tabla 4.1 se muestra de una forma concisa la analogía entre estructuras de datos y estructuras de programas.

En el Cap. 3 se vio que la iteración es un caso especial de recursión y que una llamada a un procedimiento recursivo **P** definido según el esquema (4.9)

```

procedure P;
begin
  if B then begin P0; P end
end
  
```

(4.9)

Forma constructiva	Instrucción de programa	Tipo de datos
Elemento atómico	Asignación	Tipo escalar
Enumeración	Instrucción compuesta	Tipo registro
Repetición por un factor conocido	Instrucción for	Tipo array
Elección	Instrucción condicional	Registro con variante, unión de tipos
Repetición por un factor desconocido	Instrucción while o repeat	Tipo secuencia o fichero
Recursión	Instrucción procedimiento	Tipo recursivo de datos
«Grafo» genérico	Instrucción goto	Estructura enlazada por punteros

Tabla 4.1. Correspondencia entre estructuras de datos y de programas.

donde **P₀** es una instrucción que no involucra a **P**, es equivalente a, y reemplazable por, la instrucción iterativa

while B do P₀

Las analogías mostradas en la Tabla 4.1 revelan que existe una relación similar entre tipos recursivos de datos y la secuencia. De hecho, un tipo recursivo definido según el esquema

```

type T = record
  if B then (t0: T0; t: T)
  end
  
```

(4.10)

siendo **T₀** un tipo que no involucra a **T**, es equivalente a, y reemplazable por, el tipo secuencial de datos

file of T₀

Lo que se ha visto hasta ahora muestra que la recursión puede ser reemplazada por la iteración tanto en las definiciones de datos como en las definiciones de programas si (y sólo si) el nombre del procedimiento o del tipo se utiliza recursivamente sólo una vez al final (o al principio) de su definición.

El resto de este capítulo está dedicado a la generación y manipulación de estructuras de datos cuyos componentes están enlazados por punteros explícitos. Se da atención especial a determinadas estructuras sencillas; a partir de su estudio se puede llegar a mecanismos de manipulación de estructuras más complejas. Las estructuras simples son la lista lineal o secuencia encadenada, que es el caso más sencillo, y los árboles. El especial interés dedicado a estructuras de datos tan simples no quiere decir que en la práctica no se presenten estructuras de datos más complejas. De hecho, la historia que se presenta a continuación, aparecida en un diario de Zürich en julio de 1922, es una prueba de que, incluso

en casos que sirven normalmente de ejemplos de estructuras regulares, tales como los árboles genealógicos, pueden presentarse irregularidades complejas de tratar. La historia habla de un hombre que describe los problemas de su vida en los términos siguientes:

Me casé con una viuda que tenía una hija. Mi padre, que nos visitaba muy frecuentemente, se enamoró de ella y se casaron. Por lo tanto, mi padre se convirtió en mi yerno, y mi hija se convirtió en mi madre. Unos meses más tarde, mi mujer tuvo un hijo, que se convirtió en el hermano de mi padre y también en mi tío. La mujer de mi padre, mi hija, también tuvo un hijo. Por lo tanto, tuve un hermano y al mismo tiempo un nieto. Mi mujer es mi abuela, ya que es la madre de mi madre. Por lo tanto, soy el marido de mi mujer y al mismo tiempo su nieto; en otras palabras, soy mi propio abuelo.

4.3. LISTAS LINEALES

4.3.1. Operaciones básicas

La forma más simple de relacionar o enlazar un conjunto de elementos es alinearlos formando una única *lista* o *cola* pues, en este caso, se necesita únicamente un enlace por elemento, para referenciar su sucesor.

Supóngase definido un tipo *T* en la forma indicada en (4.11). Cada variable de este tipo se compone de tres partes, es decir, una clave que la identifica, el puntero a su sucesor, y una posible información adicional asociada con la clave, que se omite en (4.11).

```
type T = record clave: integer;
           suce: ↑T;
           .....
end
```

 (4.11)

En la Fig. 4.6 se exhibe una lista de elementos de tipo *T* junto con una variable *p* que tiene asignada un puntero al primer componente de la lista. Probablemente, la operación más simple que se puede realizar con una lista del tipo que refleja la Fig. 4.6 es la *inserción de un elemento al comienzo de la misma*. Primero, se crea un elemento de tipo *T*, y su referencia (puntero) se asigna a una variable auxiliar de tipo puntero, *q*. A continuación, una simple reasignación de punteros completa la operación, tal como se ve en (4.12).

```
new(q); q↑ .suce := p; p := q
```

 (4.12)

Obsérvese que el orden en que se realizan estas tres instrucciones es esencial.

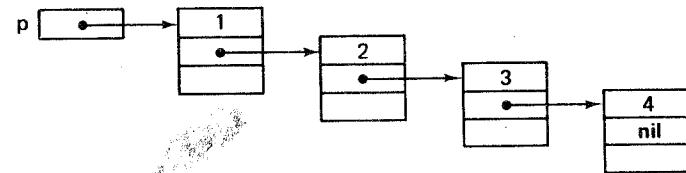


Fig. 4.6. Ejemplo de lista.

La operación de insertar un elemento al comienzo de una lista sugiere inmediatamente cómo puede generarse una lista de este tipo: comenzando con la lista vacía, se añade repetidamente un elemento al principio. El proceso de *generación de una lista* se expresa en (4.13); aquí el número de elementos a enlazar es *n*.

```
p := nil; {comenzar con la lista vacía}
while n > 0 do
  begin new(q); q↑ .suce := p; p := q;
        q↑ .clave := n; n := n - 1
  end
```

 (4.13)

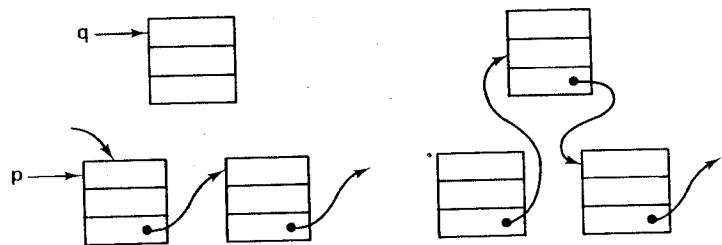
Esta es la forma más simple de crear una lista. Sin embargo, el orden de los elementos en la misma es el inverso del orden en que han «llegado». En ciertas aplicaciones esto no es deseable; consecuentemente, los nuevos elementos deben ser añadidos al *final* de la lista. Aunque el final puede encontrarse fácilmente haciendo una inspección secuencial de la lista, este método requiere un esfuerzo que puede ser ahorrado si se utiliza un segundo puntero, *q*, que apunte siempre al último elemento. Este método se aplica, por ejemplo, en el Programa 4.4, que genera referencias cruzadas de un texto dado. Tiene como inconveniente que la inserción del primer elemento debe realizarse de forma distinta a la de todos los demás. El uso explícito de punteros hace que ciertas operaciones, que de otra forma serían tediosas, puedan realizarse muy simplemente; entre las operaciones elementales que se pueden realizar con listas se encuentran las de insertar y sacar elementos (actualización selectiva de una lista), y, por supuesto, el recorrido de una lista. Se examinará primero la *inserción en listas*.

Supóngase que se desea insertar en una lista, *detrás* del elemento designado por el puntero *p*, el elemento designado por la variable puntero *q*. Las asignaciones de punteros necesarias se expresan en (4.14), y su efecto se visualiza en la Fig. 4.7.

```
q↑ .suce := p↑ .suce; p↑ .suce := q
```

 (4.14)

104 ESTRUCTURAS DINAMICAS DE INFORMACION

Fig. 4.7. Inserción en lista detrás de p^\uparrow .

Si se desea realizar la inserción antes, y no *después*, del elemento p^\uparrow , parece que la cadena de enlaces unidireccionales causa un problema, puesto que no existe ningún camino para llegar a los antecesores de un elemento. Sin embargo, un artificio simple soluciona el problema: se expresa en (4.15) y se ilustra en la Fig. 4.8. Supóngase que la clave del nuevo elemento es $c = 8$.

$$\begin{aligned} \text{new}(q); q^\uparrow := p^\uparrow; \\ p^\uparrow .\text{clave} := c; p^\uparrow .\text{suce} := q \end{aligned} \quad (4.15)$$

El «truco» consiste, evidentemente, en insertar realmente un nuevo componente detrás de p^\uparrow , intercambiando a continuación los valores del nuevo elemento y de p^\uparrow .

A continuación se considera el proceso de *borrado en listas*. Borrar el sucesor de un elemento p^\uparrow no presenta problemas. En (4.16) se muestra el proceso, junto con la reinserción del elemento borrado al comienzo de otra lista (designada por q). r es una variable auxiliar del tipo $\uparrow T$.

$$\begin{aligned} r := p^\uparrow .\text{suce}; p^\uparrow .\text{suce} := r^\uparrow .\text{suce}; \\ r^\uparrow .\text{suce} := q; q := r \end{aligned} \quad (4.16)$$

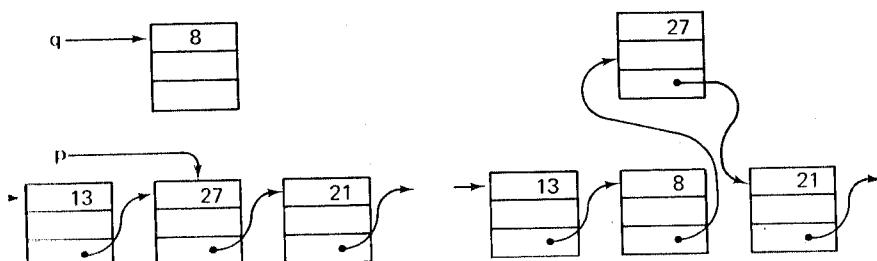
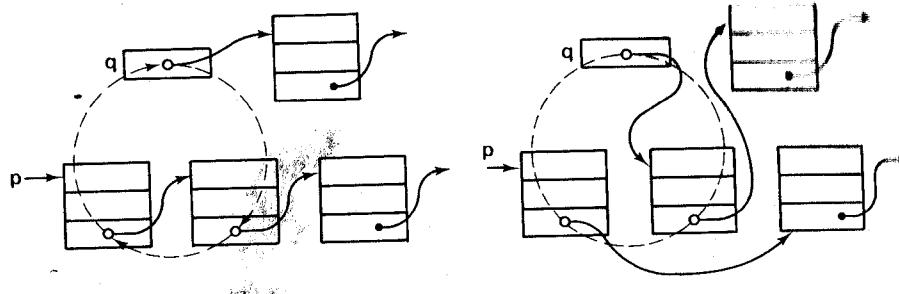
Fig. 4.8. Inserción en lista antes de p^\uparrow .

Fig. 4.9. Borrado en lista y reinserción.

La Fig. 4.9 ilustra el proceso (4.16) y muestra que consiste de un intercambio cíclico de tres punteros.

Sacar un cierto elemento (en vez de su sucesor) resulta más difícil, al presentarse el mismo problema que al insertar delante de un elemento p^\uparrow : volver al elemento anterior del dado es imposible. Pero borrar su sucesor después de haber movido su valor hacia adelante es una solución relativamente obvia y sencilla. Se puede aplicar siempre que p^\uparrow tenga un sucesor, es decir, no sea el último elemento de la lista.

A continuación se estudia la operación fundamental de *recorrido de una lista*. Supóngase que hay que realizar una operación $P(x)$ con todos los elementos de una lista cuyo primer elemento es p^\uparrow . Esta tarea puede expresarse como sigue:

```
while lista designada por  $p$  no este vacía do
  begin realizar operación  $P$ ;
    continuar con el sucesor
  end
```

La instrucción (4.17) describe esta operación en detalle.

```
while  $p \neq \text{nil}$  do
  begin  $P(p^\uparrow); p := p^\uparrow .\text{suce}$ 
  end \quad (4.17)
```

Se desprende de las definiciones de la instrucción **while** y de la estructura de enlaces que P se aplica a todos los elementos de la lista y a ningún otro.

Una operación que se realiza muy frecuentemente es la *búsqueda en una lista* de un elemento con una clave determinada x . Igual que en las estructuras fichero, la búsqueda es puramente secuencial. La búsqueda termina, bien cuando se encuentra un elemento, o bien cuando se llega al final de la lista. Supóngase de nuevo que el comienzo de la lista se designa por un puntero p . Un primer intento de formular esta búsqueda sencilla produce lo siguiente:

while ($p \neq \text{nil}$) \wedge ($p^\uparrow .clave \neq x$) **do** $p := p^\uparrow .suce$ (4.18)

Sin embargo, obsérvese que $p = \text{nil}$ implica que p^\uparrow no existe. Por lo tanto, la evaluación de la condición límite puede implicar el acceso a una variable inexistente (no ya a una variable con valor indefinido) y puede producir error en la ejecución del programa. Esto puede subsanarse, bien utilizando una interrupción explícita de la repetición mediante una sentencia **goto** (4.19), o bien introduciendo una variable booleana auxiliar que indique si una clave deseada ha sido encontrada o no (4.20).

while $p \neq \text{nil}$ **do**
if $p^\uparrow .clave = x$ **then goto** *Encontrada*
else $p := p^\uparrow .suce$ (4.19)

El uso de la sentencia **goto** requiere la presencia de una etiqueta en algún sitio; obsérvese que su incompatibilidad con la sentencia **while** se hace evidente por el hecho de que la cláusula «mientras» induce a confusión: la instrucción controlada *no* se ejecuta necesariamente mientras $p \neq \text{nil}$.

$b := \text{true};$
while ($p \neq \text{nil}$) $\wedge b$ **do**
if $p^\uparrow .clave = x$ **then** $b := \text{false}$
else $p := p^\uparrow .suce$
 $\{(p = \text{nil}) \vee \neg b\}$ (4.20)

4.3.2. Listas ordenadas y listas reorganizables

El algoritmo (4.20) se parece mucho a las rutinas de búsqueda en un array o un fichero. De hecho, un fichero no es más que una lista lineal en la cual se ha dejado sin especificar, o implícita, la técnica de enlace con el elemento siguiente. Como los operadores primitivos de ficheros no permiten la inserción de nuevos elementos (excepto al final) o el borrado de elementos (excepto el borrado de *todos* los elementos), el responsable de la implantación dispone de amplia libertad en la elección de la representación a utilizar, y puede muy bien utilizar asignación secuencial de memoria, colocando componentes sucesivos en áreas de memoria consecutivas. Las listas lineales con punteros explícitos proporcionan más flexibilidad y, por lo tanto, deben ser utilizadas siempre que se necesite esta flexibilidad adicional.

Como ejemplo, se va a considerar ahora un problema, que se presentará con frecuencia a lo largo de este capítulo, con vistas a ilustrar diversas soluciones y técnicas. El problema consiste en leer un texto, examinar todas sus palabras, y contar la frecuencia de utilización de cada una de ellas. Se llama la construcción de una *concordancia*.

Una solución obvia es construir una *lista* de las palabras encontradas en el

texto. La lista se examina palabra por palabra. Si la palabra se encuentra en la lista, se incrementa su contador de frecuencia; si no, se añade la palabra a la lista. Se denominará este proceso *buscar*, aunque puede aparentemente incluir una *inserción*.

Para poder concentrarse en la parte esencial del manejo de la lista, se supone que las palabras ya han sido extraídas del texto que se investiga, han sido codificadas como enteros, y están disponibles en forma de fichero de entrada de datos.

En base a (4.20) puede escribirse fácilmente el procedimiento llamado *buscar*. La variable *raiz* apunta al comienzo de la lista en la que se insertan nuevas palabras según (4.12). El algoritmo completo se presenta en el Programa 4.1; incluye una rutina que tabula la lista de concordancias construida. El proceso de tabulación es un ejemplo de acción que se ejecuta una vez por cada elemento de la lista, tal como se presenta de forma esquemática en (4.17).

El algoritmo de búsqueda del Programa 4.1 se parece al procedimiento de búsqueda en arrays y ficheros, y sugiere la técnica utilizada allí para simplificar la condición de terminación del bucle: el uso de un *centinela*. También puede utilizarse un centinela en la lista; se representa por un elemento ficticio al final de la lista. El nuevo procedimiento es el (4.21), que sustituye al procedimiento

Programa 4.1. Inserción simple en lista.

```
program lista (input, output);
{insercion directa en lista}
type ref = ↑palabra;
palabra = record clave: integer;
cuenta: integer;
suce: ref
end;
var k: integer; raiz: ref;
procedure buscar (x: integer; var raiz: ref);
var p: ref; b: boolean;
begin p := raiz; b := true;
while (p  $\neq$  nil)  $\wedge b$  do
if  $p^\uparrow .clave = x$  then  $b := \text{false}$  else  $p := p^\uparrow .suce$ 
if b then
begin {nuevo elemento}  $p := \text{raiz}; \text{new}(\text{raiz});$ 
with  $\text{raiz}^\uparrow$  do
begin clave := x; cuenta := 1; suce := p
end
end
end else
 $p^\uparrow .cuenta := p^\uparrow .cuenta + 1$ 
end {buscar};
```

```

procedure imprimirlista (p: ref);
begin while p ≠ nil do
  begin writeln (p↑ .clave, p↑ .cuenta);
    p := p↑ .suce
  end
end {imprimirlista};
begin raiz := nil; read(k);
while k ≠ 0 do
  begin buscar (k, raiz); read(k)
  end;
  imprimirlista(raiz)
end.

```

Programa 4.1 (Continuación)

búsqueda del Programa 4.1, suponiendo que se añade una variable global, *centinela*, y que la inicialización de *raiz* se sustituye por las instrucciones:

```
new(centinela); raiz := centinela;
```

que generan el elemento que se va a utilizar como centinela.

```

procedure buscar(x: integer; var raiz: ref);
  var p: ref;
begin p := raiz; centinela↑ .clave := x;
  while p↑ .clave ≠ x do p := p↑ .suce;
  if p ≠ centinela then p↑ .cuenta := p↑ .cuenta + 1 else
    begin {nuevo elemento} p := raiz; new(raiz);
      with raiz↑ do
        begin clave := x; cuenta := 1; suce := p
        end
    end
end {buscar}

```

(4.21)

Obviamente, la potencia y flexibilidad de la lista enlazada no están adecuadamente utilizadas en este ejemplo, y el examen de la lista completa puede ser aceptado solamente en casos en los que el número total de elementos sea pequeño. Es fácil, sin embargo, mejorar el proceso: La búsqueda en lista ordenada. Si la lista está ordenada (por ejemplo, por claves ascendentes), entonces la búsqueda puede terminarse como mucho cuando se encuentre la primera clave que sea mayor que la del nuevo elemento. Se consigue la ordenación de la lista insertando los nuevos elementos en el lugar apropiado, en vez de al comienzo de la lista. De hecho, la ordenación de la lista se obtiene prácticamente sin coste adicional. Esto es así debido a la facilidad con que se puede realizar la inserción en una lista enlazada, aprovechando su flexibilidad. Esta es una posibilidad que no tienen

las estructuras array y fichero. (Obsérvese, sin embargo, que, incluso en listas ordenadas, no existe equivalente de la búsqueda binaria en arrays.)

La búsqueda en lista ordenada es un ejemplo típico de la situación descrita en (4.15) donde un elemento debe ser insertado *delante* de un cierto ítem, a saber, delante del primero cuya clave le supere. La técnica que se utiliza a continuación, sin embargo, difiere de la empleada en (4.15). En vez de copiar valores, se usan dos punteros en el recorrido de la lista; *p₂* se mantiene un lugar detrás de *p₁* y, de esta forma, identifica el lugar apropiado para realizar la inserción, cuando *p₁* ha encontrado una clave superior. La forma de inserción se muestra en la Fig. 4.10. Antes de continuar hay que considerar dos circunstancias:

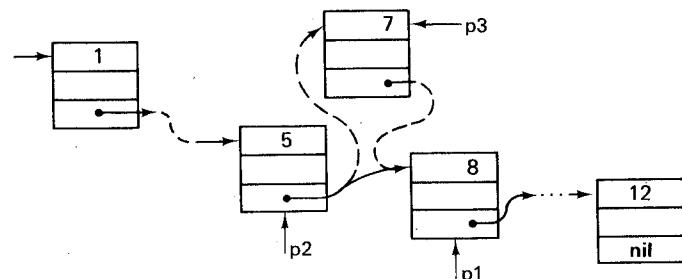


Fig. 4.10. Inserción en lista ordenada.

1. El puntero al nuevo elemento (*p₃*) debe ser asignado a *p₂↑ .suce*, excepto cuando la lista está todavía vacía. Por razones de sencillez y eficacia, se prefiere no hacer esta distinción mediante una instrucción condicional. La única forma de evitarlo es introduciendo un elemento ficticio al comienzo de la lista.
2. El examen de la lista con dos punteros que descienden a lo largo de ella, con un elemento de separación entre los mismos, requiere que la lista contenga al menos un elemento (además del ficticio). Esto hace que la inserción del primer elemento deba ser realizada de forma diferente a todos los demás.

En (4.23) se da una solución de acuerdo con las consideraciones anteriores. Se utiliza un procedimiento auxiliar, *insertar*, declarado como local respecto a *buscar*. Este procedimiento genera e inicializa el nuevo elemento *p* tal como se muestra en (4.22).

```

procedure insertar(p: ref);
  var p3: ref;
begin new(p3);
  with p3↑ do
    begin clave := x; cuenta := 1; suce := p
    end;
    p2↑ .suce := p3
end {insertar}

```

(4.22)

La instrucción de inicialización «*raiz* := nil» del Programa 4.1 ha sido sustituida por

new(raiz); raiz[↑].*suce* := nil

Refiriéndose a la Fig. 4.10, se determina la condición con la que el examen de la lista pasa al siguiente elemento; consiste de dos factores,

$$(p1^{\uparrow}.\text{clave} < x) \wedge (p1^{\uparrow}.\text{suce} \neq \text{nil})$$

En (4.23) se muestra el procedimiento de búsqueda resultante.

```
procedure buscar(x: integer; var raiz: ref);
  var p1, p2: ref;
begin p2 := raiz; p1 := p2↑.suce;
  if p1 = nil then insertar(nil) else
    begin
      while (p1↑.clave < x)  $\wedge$  (p1↑.suce  $\neq$  nil) do
        begin p2 := p1; p1 := p2↑.suce
        end;
        if p1↑.clave = x then p1↑.cuenta := p1↑.cuenta + 1 else
          insertar(p1)
      end
    end {buscar};
```

(4.23)

Desgraciadamente, esta solución contiene un fallo lógico. A pesar del cuidadoso análisis hecho, ¡se ha colado un error! El lector debe tratar de encontrarlo antes de continuar. Para aquellos que prefieran no hacer de detectives, se dirá únicamente que (4.23) siempre «empujará» el primer elemento insertado hacia el final de la lista. El fallo se corrige teniendo en cuenta que, si el examen de la lista termina debido al segundo factor, el nuevo elemento debe ser insertado *después de p1*[↑] en vez de *antes*. Por lo tanto, la instrucción «*insertar(p1)*» debe ser sustituida por

```
begin if p1↑.suce = nil then
  begin p2 := p1; p1 := nil
  end;
  insertar(p1)
end
```

(4.24)

Maliciosamente, el lector confiado ha sido engañado una vez más, pues (4.24) es todavía incorrecto. Para poner en evidencia el error, supóngase que la nueva

clave se encuentra entre la última y la penúltima. Esto dará como resultado que ambos factores en la condición de continuación sean falsos cuando el examen de la lista alcanza el final de la misma y, consecuentemente, la inserción se hace detrás del elemento que está al final. Si la misma clave se presenta de nuevo más tarde, será insertada correctamente y aparecerá entonces dos veces en la lista. El remedio consiste en reemplazar la condición

p1[↑].*suce* = nil

en (4.24) por

p1[↑].*clave* < *x*

Para acelerar la búsqueda, se puede otra vez simplificar la condición de continuación de la instrucción **while** utilizando un centinela. Esto requiere la presencia inicial de una *cabecera ficticia* así como de un centinela al final de la lista. Por lo tanto, la lista debe ser inicializada con las instrucciones siguientes

new(raiz); new(centinela); raiz[↑].*suce* := *centinela*;

y el procedimiento de búsqueda resulta considerablemente más simple, tal como se ve en (4.25).

```
procedure buscar(x: integer; var raiz: ref);
  var p1, p2, p3: ref;
begin p2 := raiz; p1 := p2↑.suce; centinela↑.clave := x;
  while p1↑.clave < x do
    begin p2 := p1; p1 := p2↑.suce
    end;
    if (p1↑.clave = x)  $\wedge$  (p1  $\neq$  centinela) then
      p1↑.cuenta := p1↑.cuenta + 1 else
    begin new(p3); {insertar p3 entre p1 y p2}
      with p3↑ do
        begin clave := x; cuenta := 1; suce := p1
        end;
        p2↑.suce := p3
      end
    end {buscar}
```

(4.25)

Ha llegado el momento de preguntarse qué ventajas pueden esperarse de la búsqueda en lista ordenada. Teniendo en cuenta que la complejidad adicional obtenida es pequeña, uno no debería esperarse una mejora excepcional.

Supóngase que todas las palabras del texto se presentan con igual frecuencia.

En este caso, la ventaja obtenida con la ordenación lexicográfica es desde luego nula, una vez que todas las palabras están en la lista, ya que la posición de una palabra no importa, si todas las palabras se presentan con la misma frecuencia; sólo el *total* de los accesos individuales a elementos de la lista es significativo. Sin embargo, se obtiene una ventaja cuando hay que insertar una nueva palabra. En vez de examinar primero la lista completa, por término medio habrá que examinar sólo la mitad de la lista. Por lo tanto, la inserción en una lista ordenada presenta ventajas sólo si se va a generar una concordancia que tenga muchas palabras distintas, que aparezcan con frecuencias bajas. Los ejemplos anteriores son, por ello, adecuados, principalmente, como ejercicios de programación, más que como aplicaciones prácticas.

La organización de datos como una lista ordenada es recomendable cuando el número de elementos es relativamente pequeño (por ejemplo < 100), varía y, además, no se tiene información sobre las frecuencias de acceso a los elementos. Un ejemplo típico es la tabla de símbolos de los compiladores de los lenguajes de programación. Cada declaración provoca la adición de un nuevo símbolo a la tabla y, una vez fuera del campo de validez del mismo, el elemento se saca de la tabla. El uso de listas encadenadas simples es apropiado en aplicaciones con programas relativamente cortos. Incluso en este caso, se puede obtener una mejora considerable en el método de acceso, utilizando una técnica sencilla que se menciona aquí de nuevo, principalmente porque constituye un buen ejemplo para mostrar la flexibilidad de la estructura de lista encadenada.

Una propiedad característica de los programas es que los puntos donde se presenta un mismo identificador suelen estar agrupados, es decir, el punto donde se presenta una palabra está normalmente seguido por uno o más puntos donde se presenta la misma palabra. Esta información es una invitación para reorganizar la lista, después de cada acceso, moviendo la palabra que se ha encontrado al comienzo de la lista, y minimizando por tanto la longitud de la búsqueda la próxima vez que se busque la misma palabra. Este método de acceso se llama *búsqueda en lista con reordenación*, o —un poco pomposamente— *búsqueda en lista auto-organizable*. Al presentar el algoritmo correspondiente en forma de un procedimiento, que puede ser colocado en el Programa 4.1, se aprovecha la experiencia adquirida hasta el momento y se introduce un centinela inmediatamente. De hecho, un centinela no sólo hace la búsqueda más rápida, sino que, en este caso, también simplifica el programa. La lista está, sin embargo, no vacía al principio, sino que contiene ya el elemento centinela. Las instrucciones iniciales son

```
new (centinela); raiz := centinela;
```

Obsérvese que la principal diferencia entre el nuevo algoritmo y la búsqueda simple en lista (4.21) es la acción de reordenar cuando se ha encontrado un elemento. Es retirado entonces de su posición anterior e insertado al principio. Esto requiere nuevamente el uso de dos punteros consecutivos, de forma que el predecesor,

p_2 , de un elemento identificado, $p_1 \uparrow$, siga siendo localizable. Esto, a su vez, requiere el tratamiento especial del primer elemento (es decir, la lista vacía). Para entender el proceso de cambio de enlaces, véase la Fig. 4.11. Esta figura muestra los dos punteros cuando $p_1 \uparrow$ ha sido identificado como el elemento deseado. La Fig. 4.12 representa la nueva configuración de la lista después de hacer la reordenación correcta, y el nuevo procedimiento de búsqueda se muestra en (4.26).

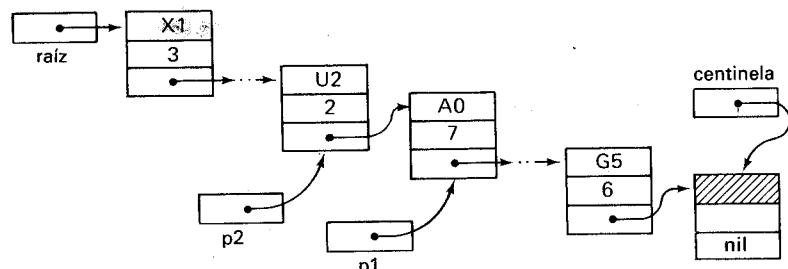


Fig. 4.11. Lista antes de reordenar.

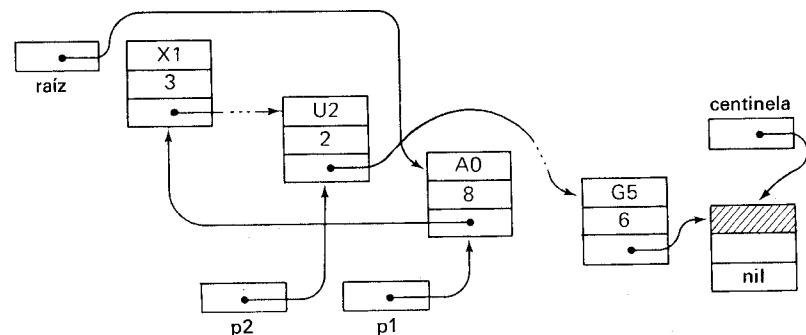


Fig. 4.12. Lista después de reordenar.

```

procedure buscar(x: integer; var raiz: ref);
  var p1, p2: ref;
begin p1 := raiz; centinela↑ .clave := x;
  if p1 = centinela then
    begin {primer elemento} new(raiz);
      with raiz↑ do
        begin clave := x; cuenta := 1; suce := centinela
        end
    end
  else
    
```

(4.26)

```

if  $p1 \uparrow .clave = x$  then  $p1 \uparrow .cuenta := p1 \uparrow .cuenta + 1$  else
begin {buscar}
repeat  $p2 := p1; p1 := p2 \uparrow .suce$ 
until  $p1 \uparrow .clave = x;$ 
if  $p1 = centinela$  then
begin {insertar}
 $p2 := raiz; new(raiz);$ 
with  $raiz \uparrow$  do
begin  $clave := x; cuenta := 1; suce := p2$ 
end
end else
begin {encontrado, ahora reordenar}
 $p1 \uparrow .cuenta := p1 \uparrow .cuenta + 1;$ 
 $p2 \uparrow .suce := p1 \uparrow .suce; p1 \uparrow .suce := raiz; raiz := p1$ 
end
end
end {buscar}

```

(4.26)

La ventaja obtenida con este método de búsqueda depende en gran medida del grado de agrupamiento de los datos de entrada. Para un factor de agrupamiento dado, la ventaja obtenida será más importante en listas grandes. Para dar una idea del orden de magnitud de la ventaja que se puede esperar, se ha hecho una medición empírica, aplicando el programa de concordancia anterior a un texto corto y a un texto relativamente largo, y se han comparado los métodos de lista lineal ordenada (4.21) y de reorganización de lista (4.26). Los datos medidos se presentan en la Tabla 4.2. Desgraciadamente, la ventaja es mayor cuando, de todas formas, lo que se necesita es un tipo distinto de organización de los datos. Se volverá a este mismo ejemplo en el apartado 4.4.

	Test 1	Test 2
Número de claves distintas	53	582
Número de claves totales	315	14341
Tiempo de búsqueda con ordenación	6207	3200622
Tiempo de búsqueda con reorganización	4529	681584
Factor de mejora	1,37	4,70

Tabla 4.2. Comparación de métodos de búsqueda en lista.

4.3.3. Una aplicación: ordenación topológica

Un ejemplo adecuado de uso de una estructura de datos dinámica, flexible, es el proceso de *ordenación topológica*. Este es un proceso de ordenación aplicable a elementos en los que está definido un *orden parcial*, es decir, existe un orden entre ciertos pares de elementos, pero no entre todos ellos. Esta es una situación que se presenta con frecuencia. A continuación se dan algunos ejemplos de órdenes parciales:

1. En un diccionario o glosario, las palabras se definen en función de otras palabras. Si una palabra v está definida en términos de una palabra w , se dice que $v \prec w$. La ordenación topológica de las palabras en un diccionario significa organizarlas de tal manera que todas las palabras que se utilizan en la definición de una dada hayan sido definidas previamente.
2. Una tarea (por ejemplo, un proyecto de ingeniería) se divide en subtareas. Normalmente, la ejecución de ciertas subtareas debe estar precedida por la terminación de otras. Si la ejecución de una subtarea v debe preceder a la de una subtarea w , se escribe $v \prec w$. La ordenación topológica significa organizar las subtareas de tal forma que, al iniciar una subtarea, todas aquéllas que son requisitos de la misma hayan sido ya realizadas.
3. En una carrera universitaria, ciertas asignaturas deben ser aprobadas antes de matricularse de otras. Si una asignatura v es un requisito de una asignatura w , se escribe $v \prec w$. La ordenación topológica significa ordenar las asignaturas de tal forma que ninguna que sea requisito de otra venga a continuación de ella.
4. En un programa, algunos procedimientos pueden tener llamadas a otros procedimientos. Si un procedimiento v es llamado por un procedimiento w , se escribe $v \prec w$. La ordenación topológica implica organizar las declaraciones de los procedimientos de tal forma que en la declaración de cada uno de ellos no exista ninguna referencia a otro declarado posteriormente.

En general, un orden parcial de un conjunto S es una relación entre los elementos de S . Se designa por el símbolo \prec , que se lee como «precede», y satisface las siguientes tres propiedades (axiomas), dados tres elementos distintos cualesquiera de S , x , y , z :

- (1) si $x \prec y$ y $y \prec z$, entonces $x \prec z$ (transitiva)
 - (2) si $x \prec y$, entonces no $y \prec x$ (antisimétrica)
 - (3) no $x \prec x$ (irreflexiva)
- (4.27)

Por razones evidentes, se supondrá que los conjuntos S que van a ser ordenados topológicamente por un algoritmo son finitos. Entonces, un orden parcial puede ser ilustrado por medio de un diagrama o grafo en el cual los nodos representan los elementos de S y las flechas representan relaciones de orden entre ellos. En la Fig. 4.13 aparece un ejemplo.

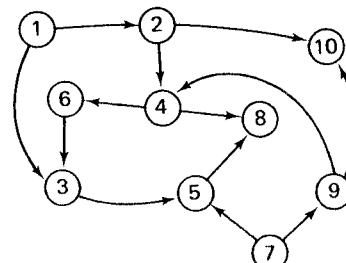


Fig. 4.13. Conjunto parcialmente ordenado.

El problema de la ordenación topológica consiste en encontrar un orden lineal que comprenda al orden parcial. Gráficamente, esto implica la organización de los nodos del grafo en una fila, de tal forma que todas las flechas señalen a la derecha, tal como se muestra en la Fig. 4.14. Las propiedades (1) y (2) de los órdenes parciales garantizan que el grafo no contiene ciclos. Esta es precisamente la condición necesaria para poder encontrar tal orden lineal.

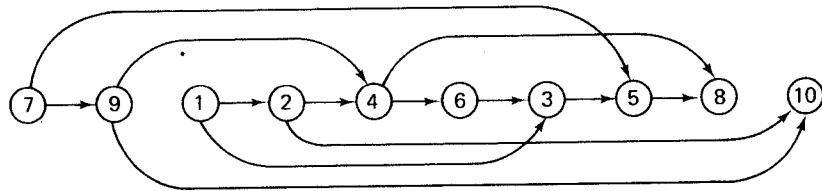


Fig. 4.14. Organización lineal del conjunto parcialmente ordenado de la Fig. 4.13.

¿Cómo se actúa para encontrar uno de los posibles órdenes lineales? La fórmula es bien sencilla. Se empieza eligiendo un elemento cualquiera que no esté precedido por ningún otro (existe al menos uno; en caso contrario habría un ciclo). Este elemento se coloca al comienzo de la lista resultante y se saca del conjunto S . El conjunto restante sigue estando parcialmente ordenado y, por lo tanto, se puede aplicar el mismo algoritmo hasta que el conjunto se vacíe.

Para poder describir este algoritmo más rigurosamente, es necesario fijar una estructura de datos y una representación de S y su orden parcial. La elección de esta representación viene determinada por las operaciones a realizar, en particular, la operación de seleccionar elementos sin ningún predecesor. Por ello, cada elemento debería representarse por tres características: su clave de identificación, su conjunto de sucesores, y un contador de precedentes. Puesto que el número n de elementos en S no se conoce *a priori*, el conjunto puede organizarse convenientemente como una lista enlazada. De acuerdo con ello, un campo adicional en la descripción de cada elemento contiene el enlace al siguiente en la lista. Se supondrá que las claves son números enteros (pero no necesariamente los enteros consecutivos del 1 al n). De manera análoga, el conjunto de sucesores de cada elemento puede representarse convenientemente como una lista enlazada. Cada elemento de la lista de sucesores se describe por una identificación y un enlace al siguiente elemento en esta lista. Llamando *líder* a un descriptor de la lista principal, en la cual cada elemento se presenta exactamente una vez, y *trailer* al descriptor de elementos en las cadenas de sus sucesores, se obtienen las siguientes declaraciones de tipos de datos:

```
type lref = ↑líder;
        tref = ↑trailer;
        líder record clave, cuenta: integer;
                    trail: tref;
```

```
suce: lref
end;
trailer = record id: lref;
           suce: tref
end
```

(4.28)

Supóngase que el conjunto S y su relación de orden están representados inicialmente como una secuencia de pares de claves en el fichero de entrada de datos. Los datos del ejemplo de la Fig. 4.13 se muestran en (4.29) donde se han añadido los símbolos \prec para mayor claridad.

1 < 2	2 < 4	4 < 6	2 < 10	4 < 8	6 < 3	1 < 3
3 < 5	5 < 8	7 < 5	7 < 9	9 < 4	9 < 10	

(4.29)

La primera parte del programa de ordenación topológica debe leer el fichero de entrada y transformar los datos en una estructura de lista. Esto se realiza leyendo sucesivamente pares de claves x e y ($x \prec y$). Supóngase que p y q designan los punteros a sus representaciones en la lista enlazada de «líderes». Estos registros deben encontrarse mediante búsqueda en la lista y , si no están aún en ella, insertarse en la misma. Esta tarea se realiza por un procedimiento tipo función llamado E . A continuación, se debe añadir un nuevo elemento, identificado con y , a la lista de «trailers» de x ; el contador de precedentes de y se aumenta en 1. Este algoritmo se llama *fase de lectura* (4.30). La Fig. 4.15 ilustra la estructura de datos generada al procesar (4.30) los datos de entrada (4.29). El programa utiliza una función $E(p)$ que produce la referencia al componente de la lista que tiene de clave p (ver también el Programa 4.2). Se supone que la secuencia de pares de entrada se termina con un cero adicional.

```
{fase de lectura} read(x);
new(cabeza); cola := cabeza; z := 0;
while x ≠ 0 do
begin read(y); p := E(x); q := E(y);
new(t); t↑ .id := q; t↑ .suce := p↑ .trail;
p↑ .trail := t; q↑ .cuenta := q↑ .cuenta + 1;
read(x)
end
```

(4.30)

Después de construir la estructura de datos de la Fig. 4.15 en la fase de lectura, se puede realizar el proceso de ordenación topológica en la forma ya descrita. Como éste consiste en seleccionar necesariamente un elemento sin ningún predecesor, parece razonable reunir primero todos los elementos que cumplen tal característica en forma de una lista enlazada. Puesto que la cadena de líderes original ya no será necesaria después, se puede reutilizar el mismo campo *suce* para enlazar los elementos líderes sin ningún predecesor. Esta operación de sus-

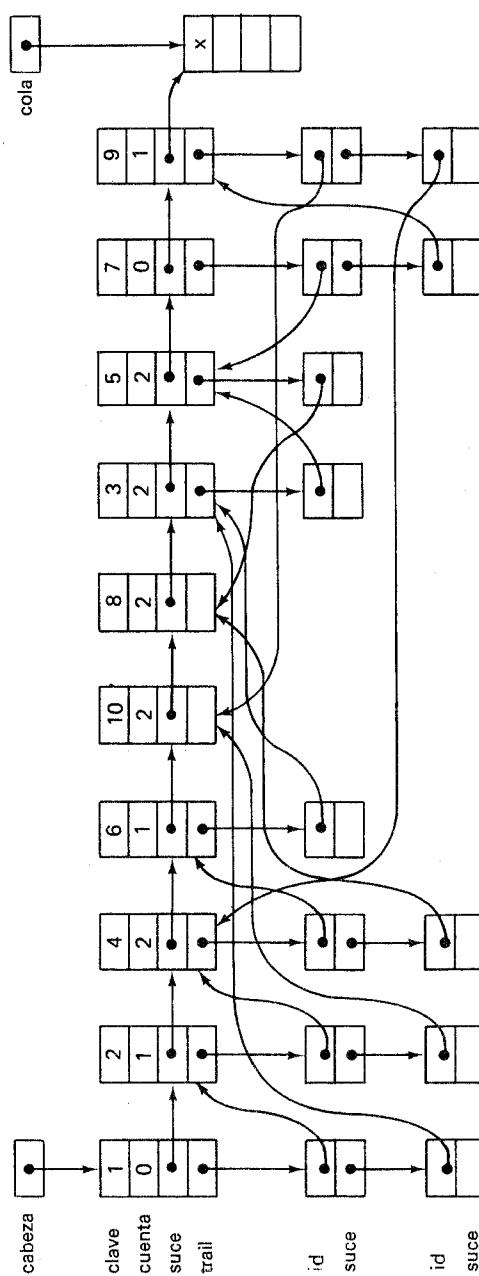


Fig. 4.15. Estructura de lista generada por el programa ordtopo.

tituir una cadena por otra se presenta a menudo en procesamiento de listas. La operación se describe en detalle en (4.31) y, para simplificar el proceso, la nueva cadena se forma en *sentido inverso*.

```
{buscar líderes sin ningún predecessor}
p := cabeza; cabeza := nil;
while p ≠ cola do
begin q := p; p := q↑ .suce;
  if q↑ .cuenta = 0 then
begin {insertar q↑ en la nueva cadena}
  q↑ .suce := cabeza; cabeza := q
end
end
```

(4.31)

La cadena de líderes *suce* de la Fig. 4.15 es reemplazada por la cadena de la Fig. 4.16 en la cual no se han dibujado los punteros que no cambian.

Después de estos trabajos preparatorios sobre la forma apropiada de representar el conjunto parcialmente ordenado *S*, se puede acometer, por último, el proceso de ordenación topológica en sí, es decir, de generar la secuencia de salida del programa. En una primera versión, éste se puede describir como sigue:

```
q := cabeza;
while q ≠ nil do
begin {escribir este elemento y eliminarlo}
  writeln(q↑ .clave); z := z - 1;
  t := q↑ .trail; q := q↑ .suce;
  «disminuir la cuenta de precedentes de todos sus sucesores en la
  lista trailer t; si alguna cuenta se hace 0, insertar este elemento
  en la lista de líderes q»
end
```

(4.32)

La instrucción a detallar en (4.32) representa otro examen secuencial de una lista [ver el esquema (4.17)]. La variable auxiliar *p* designa, en cada paso, el elemento líder cuya cuenta ha de ser disminuida y comprobada [(4.33)].

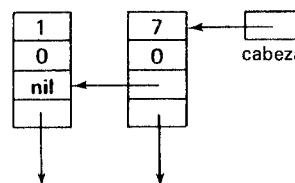


Fig. 4.16. Lista de líderes con cuenta nula.

```

while  $t \neq \text{nil}$  do
  begin  $p := t^\uparrow .id$ ;  $p^\uparrow .cuenta := p^\uparrow .cuenta - 1$ ;
    if  $p^\uparrow .cuenta = 0$  then
      begin {insertar  $p^\uparrow$  en la lista de líderes}
         $p^\uparrow .suce := q$ ;  $q := p$ 
      end;
     $t := t^\uparrow .suce$ 
  end

(4.33)

```

Con esto último se completa el programa de ordenación topológica. Obsérvese que se ha introducido un contador z , que cuenta el número de líderes que se generan en la fase de lectura. Este contador se disminuye cada vez que se escribe un líder en la fase de escritura. Por lo tanto, debería valer 0 al final del programa. Si esto no es así, se tiene una indicación de que quedan elementos en S cuando

Programa 4.2. Ordenación topológica.

```

program ordtopo(input, output);
type lref =  $\uparrow$ lader;
  tref =  $\uparrow$ trailer;
  lader = record clave: integer;
    cuenta: integer;
    trailer: tref;
    suce: lref;
  end;
  trailer = record id: lref;
    suce: tref
  end;
var cabeza, cola, p, q: lref;
  t: tref; z: integer;
  x, y: integer;
function E(p: integer): lref;
  {referencia al líder de clave p}
  var c: lref;
begin c := cabeza; cola $\uparrow .clave := p$ ; {centinela}
  while c $\uparrow .clave \neq p$  do c := c $\uparrow .suce$ ;
  if c = cola then
    begin {no hay ningún elemento con clave p en la lista}
      new(cola); z := z + 1;
      c $\uparrow .cuenta := 0$ ; c $\uparrow .trailer := \text{nil}$ ; c $\uparrow .suce := \text{cola}$ 
    end;
  E := c
end {E};

```

```

begin {inicializar la lista de líderes con un elemento ficticio}
  new(cabeza); cola := cabeza; z := 0;
  {fase de lectura} read(x);
  while x  $\neq 0$  do
    begin read(y); writeln(x, y);
      p := E(x); q := E(y);
      new(t); t $\uparrow .id := q$ ; t $\uparrow .suce := p^\uparrow .trailer$ ;
      p $\uparrow .trailer := t$ ; q $\uparrow .cuenta := q^\uparrow .cuenta + 1$ ;
      read(x)
    end;
  {buscar líderes con cuenta = 0}
  p := cabeza; cabeza := nil;
  while p  $\neq \text{nil}$  do
    begin q := p; p := p $\uparrow .suce$ ;
      if q $\uparrow .cuenta = 0$  then
        begin q $\uparrow .suce := \text{cabeza}$ ; cabeza := q
        end
    end;
  {fase de escritura} q := cabeza;
  while q  $\neq \text{nil}$  do
    begin writeln(q $\uparrow .clave$ ); z := z - 1;
      t := q $\uparrow .trailer$ ; q := q $\uparrow .suce$ ;
      while t  $\neq \text{nil}$  do
        begin p := t $\uparrow .id$ ; p $\uparrow .cuenta := p^\uparrow .cuenta - 1$ ;
          if p $\uparrow .cuenta = 0$  then
            begin {insertar  $p^\uparrow$  en la lista q}
              p $\uparrow .suce := q$ ; q := p
            end;
          t := t $\uparrow .suce$ 
        end;
      end;
      if z  $\neq 0$  then writeln ('ESTE CONJUNTO NO ESTA PARCIALMENTE ORDENADO')
    end.

```

Programa 4.2. (Continuación)

ya no hay ninguno sin ningún predecesor. Evidentemente, en este caso el conjunto S no está parcialmente ordenado.

La fase de escritura antedescrita es un ejemplo de proceso que manipula una lista que «vibra», es decir, en la cual se insertan y sacan elementos en un orden imprevisible. Es, por lo tanto, un ejemplo de proceso que utiliza todo el potencial de flexibilidad de las listas enlazadas explícitamente.

4.4. ESTRUCTURAS ARBOL

4.4.1. Definiciones y conceptos básicos

Se ha visto que las listas y secuencias pueden ser definidas convenientemente de la siguiente forma: Una secuencia (lista) de tipo base T es,

1. Bien la secuencia (lista) vacía.
2. O bien la concatenación (cadena) de un elemento de tipo T y una secuencia de tipo base T .

Aquí se ha utilizado la recursión para definir un principio de estructuración que es la iteración o yuxtaposición. Las secuencias y las iteraciones son tan comunes que se les considera normalmente formas de estructura y de comportamiento fundamentales. Pero hay que tener presente que *pueden* ser definidas en términos de recursión, mientras que la inversa no es cierta, ya que la recursión puede ser utilizada de manera elegante y efectiva para definir estructuras mucho más sofisticadas. Los árboles son un buen ejemplo de ello. Se puede definir una estructura árbol como sigue: Una *estructura árbol* con tipo base T es,

1. Bien la estructura vacía,
2. O bien un nodo de tipo T junto con un número finito de estructuras árbol, de tipo base T , disjuntas, llamadas *subárboles*.

Resulta evidente, a partir de las definiciones recursivas de las estructuras árbol y secuencia, que la secuencia (lista) es una estructura árbol en la cual cada nodo tiene como máximo un «subárbol». La secuencia (lista) es por ello conocida también por el nombre de *árbol degenerado*.

Hay varias formas de representar una estructura árbol. Por ejemplo, sea el tipo base T el conjunto de las letras del alfabeto; tal estructura árbol se presenta de diversas formas en la Fig. 4.17. Todas estas formas de representación tienen la misma estructura y son, por lo tanto, equivalentes. La representación en forma de grafo es la que exhibe explícitamente las relaciones de tipo «rama» entre los nodos y es la que ha originado, por razones obvias, el término «árbol» con el que generalmente se denomina esta estructura. Aunque resulte extraño, los árboles se dibujan normalmente dados la vuelta o, dicho de otra forma, con la raíz al aire. Esta representación, sin embargo, es confusa, pues el nodo superior (A) se denomina comúnmente *la raíz*. Aunque se debe reconocer que los árboles en la naturaleza son creaciones bastante más complejas que las abstracciones que aquí se consideran, a partir de este momento las estructuras árbol se llamarán simplemente *árboles*.

Un *árbol ordenado* es aquél en que las ramas de cada nodo están ordenadas. Por lo tanto, los dos árboles ordenados de la Fig. 4.18 son objetos diferentes. Un nodo y que está inmediatamente debajo de un nodo x se llama un *descendiente*

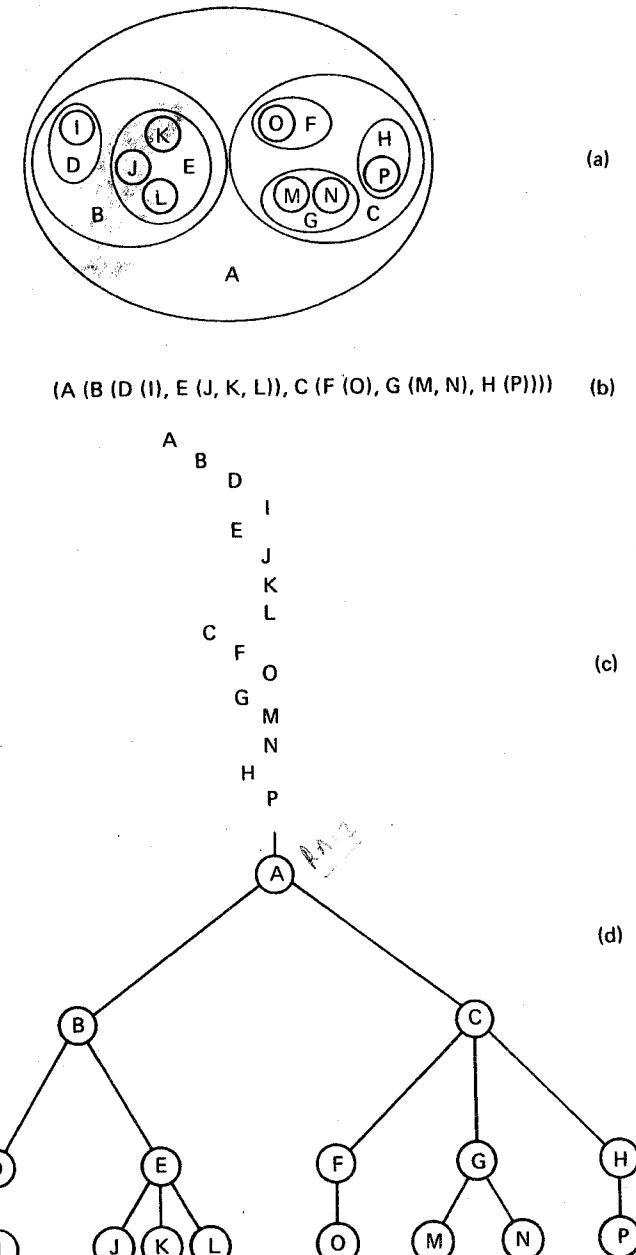


Fig. 4.17. Representación de una estructura árbol: (a) Conjuntos incluidos; (b) Paréntesis incluidos; (c) Jerarquización de márgenes; (d) Grafo.

(directo) de x ; si x está en el *nivel i*, entonces se dice que y está en el nivel $i + 1$. De forma inversa, se dice que el nodo x es el *antecesor** (directo) de y . Por definición, la raíz de un árbol está en el nivel 1. El máximo de los niveles de todos los elementos de un árbol se dice que es su *profundidad o altura*.

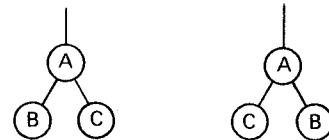


Fig. 4.18. Dos árboles binarios diferentes.

Si un elemento no tiene descendientes, se le llama *elemento terminal u hoja*; y a un elemento que no es terminal se le llama *nodo interior*. Al número de descendientes (directos) de un nodo interior se le llama su *grado*. El máximo de los grados de todos los nodos de un árbol es el *grado del árbol*. Al número de arcos que deben ser recorridos para llegar a un nodo x , partiendo de la raíz, se le llama la *longitud de camino de x*. La raíz tiene longitud de camino 1, sus descendientes directos tienen longitud de camino 2, etc. De forma general, un nodo en el nivel i tiene longitud de camino i . La longitud de camino de un árbol se define como la suma de las longitudes de camino de todos sus componentes. También se conoce con el nombre de *longitud de camino interno*. Por ejemplo, la longitud de camino interno del árbol de la Fig. 4.17 es 52. Evidentemente, la longitud de camino media C_I es

$$C_I = \frac{1}{n} \sum_i n_i \cdot i \quad (4.34)$$

siendo n_i el número de nodos en el nivel i . Para definir lo que se llama longitud de camino externo, se extiende el árbol con un nodo especial en todos los lugares del mismo donde existe un subárbol nulo. Al hacerlo, se supone que todos los nodos tienen el mismo grado y que éste es el grado del árbol. Extender el árbol de tal manera es equivalente, por lo tanto, a llenar las ramas vacías de tal forma que los nodos especiales, por supuesto, no tengan a su vez descendientes. El árbol de la Fig. 4.17, extendido con nodos especiales, se muestra en la Fig. 4.19, donde los nodos especiales están representados por cuadrados.

La *longitud de camino externo* se define en base a lo anterior como la suma de las longitudes de camino de todos los nodos especiales. Si el número de nodos especiales en el nivel i es m_i , entonces la longitud de camino externo media C_E es

$$C_E = \frac{1}{m} \sum_i m_i \cdot i \quad (4.35)$$

* N. del T.: Equivalentemente, se utilizan los nombres precedente y antecedente en la terminología usual de grafos. En esta traducción se respeta la terminología «genetológica» utilizada por el autor.

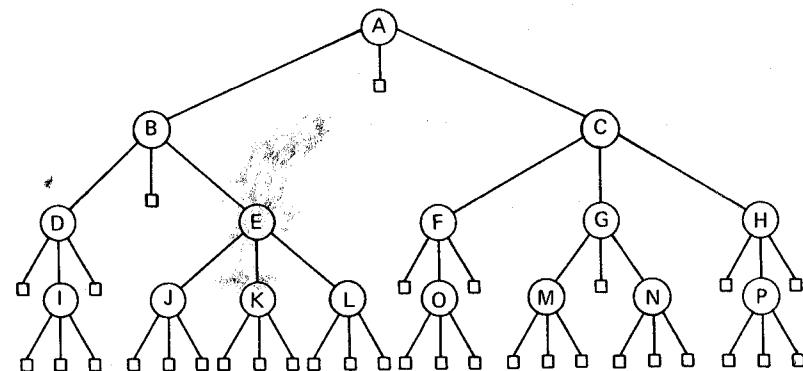


Fig. 4.19. Árbol ternario extendido con nodos especiales.

En el árbol de la Fig. 4.19 la longitud de camino externo es 153.

El número de nodos especiales m , que deben ser añadidos en un árbol de grado g , depende del número de nodos originales n . Obsérvese que un nodo cualquiera tiene exactamente un arco dirigido hacia él. Por lo tanto, existen $m + n$ arcos en el árbol extendido. Por otro lado, hay g arcos que salen de cada nodo existente inicialmente, y no sale ninguno de los nodos especiales. Así pues, existen $gn + 1$ arcos; el 1 resulta de tener en cuenta el arco que se supone dirigido hacia la raíz. Los dos resultados anteriores dan la siguiente ecuación entre el número de nodos especiales m y el de nodos originales n : $gn + 1 = m + n$, o

$$m = (g - 1)n + 1 \quad (4.36)$$

El máximo número de nodos de un árbol con una cierta altura a se alcanza cuando todos los nodos tienen g subárboles, excepto los que están en el nivel a , que no tienen ninguno. Para un árbol de grado g , el nivel 1 tiene entonces un nodo (la raíz), el nivel 2 comprende sus g descendientes, el nivel 3 los g^2 descendientes de los g nodos del nivel 2, etc. Esto da lugar a

$$N_g(a) = 1 + g + g^2 + \cdots + g^{a-1} = \sum_{i=0}^{a-1} g^i \quad (4.37)$$

nodos, como máximo, en un árbol de altura a y grado g . Para $g = 2$, se obtiene

$$N_2(a) = \sum_{i=0}^{a-1} 2^i = 2^a - 1 \quad (4.38)$$

Los árboles ordenados de grado 2 tienen una especial importancia. Se les conoce con el nombre de *árboles binarios*. Se define un árbol binario ordenado como un conjunto finito de elementos (nodos) que bien está vacío o está formado

por una raíz (nodo) con dos árboles binarios disjuntos, llamados subárbol izquierdo y derecho de la raíz.

En los apartados que siguen se considerarán únicamente árboles binarios y, por lo tanto, se utilizará la palabra «árbol» para referirse a «árbol binario ordenado». Los árboles de grado superior a 2 reciben el nombre de árboles «multicamino» y se tratan en el apartado 5 de este capítulo.

Ejemplos típicos de árboles *binarios* son el árbol genealógico, con el padre y la madre de una persona como sus descendientes (!), la historia de un campeonato de tenis, representando cada juego por un nodo con el nombre del vencedor del mismo, y los dos juegos anteriores de sus dos jugadores como sus descendientes, o una expresión aritmética formada por operadores binarios, en donde cada operador designa un nodo interno con sus operandos como subárboles (ver Fig. 4.20).

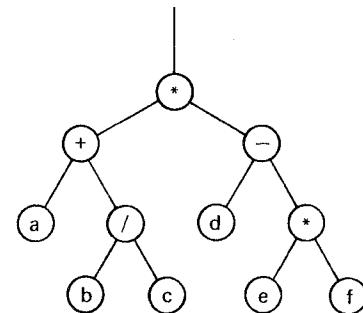


Fig. 4.20. Representación en árbol de la expresión $(a + b/c)*(d - e*f)$.

Se estudia a continuación el problema de la representación de los árboles. Está claro que el dibujo de tales estructuras utilizando arcos sugiere inmediatamente el uso de punteros. Es evidente que no tiene utilidad declarar variables con una estructura fija de árbol; en vez de ello, se definen los *nodos* como variables con una estructura fija, es decir, de un tipo determinado, en la cual el grado del árbol determina el número de componentes puntero que refieren los subárboles del nodo.

Evidentemente, la referencia al árbol vacío se denomina **nil**. Por lo tanto, el árbol de la Fig. 4.20 está formado por componentes de un tipo definido como

```
type nodo = record op: char;
    izquierdo, derecho: ^nodo
end
```

(4.39)

y puede construirse como se indica en la Fig. 4.21.

Está claro que existen formas de representar la idea abstracta de estructura árbol en base a otros tipos de datos, los arrays por ejemplo. Esto es normal hacerlo en lenguajes en que no pueden crearse componentes dinámicamente, y referenciarlos por medio de punteros. En este caso, el árbol de la Fig. 4.20 podría estar representado por una variable array declarada en la forma

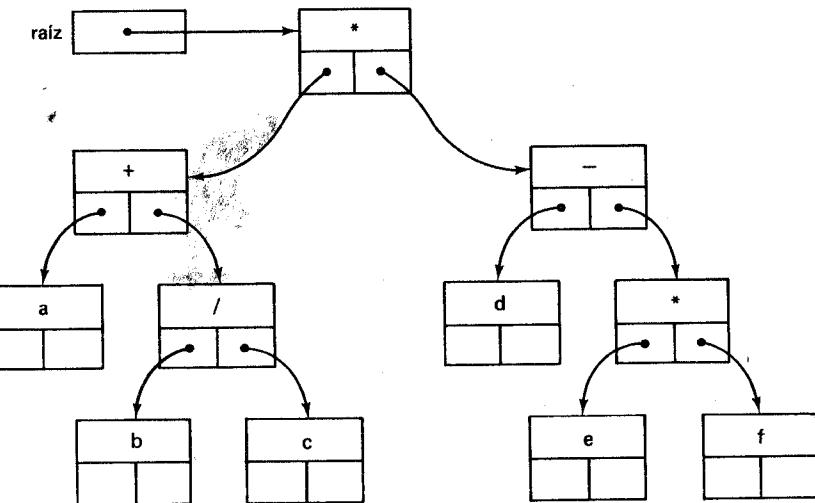


Fig. 4.21. Un árbol representado como estructura de datos.

```
a: array [1 .. 11] of
record op: char;
    izquierdo, derecho: integer
end
```

(4.40)

y con valores de los componentes tal como se muestra en la Tabla 4.3.

1	*	2	3
2	+	6	4
3	-	9	5
4	/	7	8
5	*	10	11
6	a	0	0
7	b	0	0
8	c	0	0
9	d	0	0
10	e	0	0
11	f	0	0

Tabla 4.3. Árbol representado por un array.

Aunque la estructura abstracta, subyacente, de los datos representados por el array *a* es un árbol, no se llamará a esto un árbol sino más bien un array, de acuer-

do con su declaración explícita. No se analizarán otras posibles formas de representar árboles en sistemas que carezcan de asignación dinámica de memoria, pues se supone que sistemas de programación y lenguajes que incluyen esta posibilidad son, o serán, usuales.

Antes de estudiar la forma de aprovechar las ventajas de las estructuras árbol, y la forma de realizar operaciones con ellos, se da un ejemplo de construcción de un árbol mediante un programa. Supóngase que se desea generar un árbol cuyos nodos sean del tipo definido en (4.39), siendo los valores de los nodos n números que se leen de un fichero de entrada de datos. Para hacer el problema más interesante, sea el árbol a construir aquél de n nodos que tenga altura mínima.

Para conseguir la mínima altura con un número dado de nodos, hay que colocar el máximo número posible de ellos en cada nivel excepto en el nivel más bajo. Esto puede realizarse fácilmente distribuyendo los nodos, según se lean, equitativamente a la izquierda y a la derecha de cada nodo. Esto implica estructurar el árbol, para una n dada, tal como se muestra en la Fig. 4.22 para valores de $n = 1, \dots, 7$.

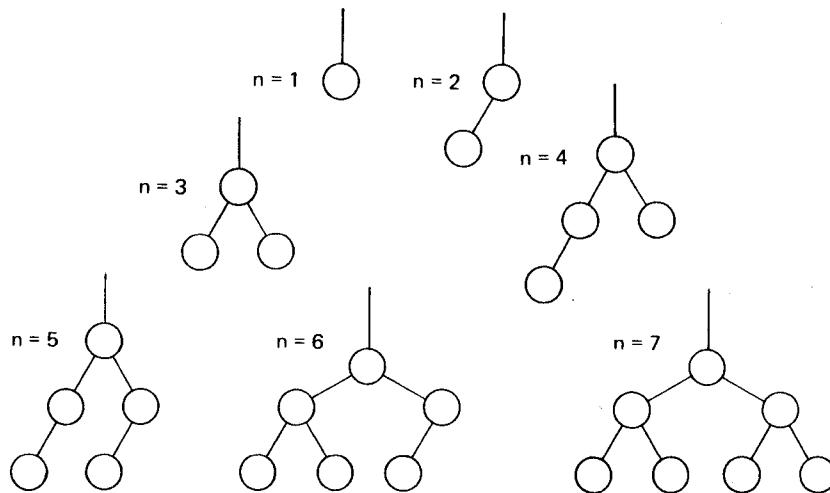


Fig. 4.22. Arboles perfectamente equilibrados.

La forma recursiva es la mejor para expresar esta regla de distribución equitativa de nodos para un número dado de ellos n :

1. Usar un nodo para la raíz.
2. Generar el subárbol izquierdo con $ni = n \text{ div } 2$ nodos utilizando la misma regla.
3. Generar el subárbol derecho con $nd = n - ni - 1$ nodos utilizando la misma regla.

La regla está expresada como un procedimiento recursivo, formando parte del Programa 4.3; este lee el fichero de entrada y construye el árbol perfectamente equilibrado. Nótese la siguiente definición:

Un árbol está *perfectamente equilibrado* si, para cada nodo, el número de nodos en el subárbol izquierdo, y el número de nodos en el subárbol derecho, difieren como mucho en una unidad.

Programa 4.3. Construcción de un árbol perfectamente equilibrado.

```

program construirarbol(input, output);
type ref = ↑nodo;
      nodo = record clave: integer;
                  izquierdo, derecho: ref
              end;
var n: integer; raiz: ref;
function arbol (n: integer): ref;
    var nuevonodo: ref;
        x, ni, nd: integer;
begin {construir arbol de n nodos perfectamente equilibrado}
    if n = 0 then arbol := nil else
    begin ni := n div 2; nd := n - ni - 1;
        read(x); new(nuevonodo);
        with nuevonodo↑ do
            begin clave := x; izquierdo := arbol(ni); derecho := arbol(nd)
            end;
        arbol := nuevonodo
    end
end {arbol};
procedure imprimirarbol(a: ref; h: integer);
    var i: integer;
begin {imprimir arbol a con margen h}
    if a ≠ nil then
        with a↑ do
            begin imprimirarbol(izquierdo, h + 1);
                for i := 1 to h do write(" ");
                writeln(clave);
                imprimirarbol(derecho, h + 1)
            end
    end {imprimirarbol};
begin {el primer entero es el numero de nodos}
    read(n);
    raiz := arbol(n);
    imprimirarbol(raiz, 0)
end.

```

Supóngase, por ejemplo, que se tienen los siguientes datos de entrada, para un árbol de 21 nodos.

```
21 8 9 11 15 19 20 21 7 3 2 1 5 6 4 13 14
10 12 17 16 18
```

El Programa 4.3 construye entonces el árbol perfectamente equilibrado de la Fig. 4.23.

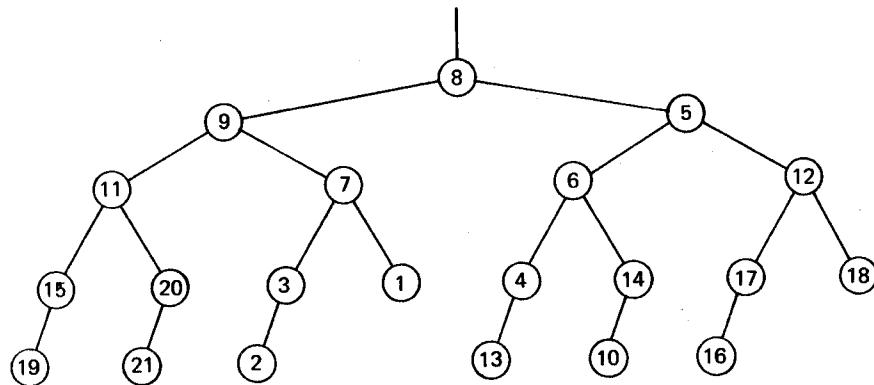


Fig. 4.23. Arbol generado por el Programa 4.3.

Obsérvese la simplicidad y transparencia de este programa, gracias a la utilización de procedimientos recursivos. Evidentemente, los algoritmos recursivos están especialmente indicados cuando un programa tiene que manipular información cuya estructura está definida a su vez de forma recursiva. Esto se comprueba nuevamente en el procedimiento que imprime el árbol resultante: el árbol prueba nuevamente en el procedimiento que imprime el subárbol en el nivel N hace que se imprima primero su propio subárbol izquierdo, a continuación el nodo, con un margen apropiadamente precedido de N espacios en blanco y, por último, el subárbol derecho.

La ventaja del algoritmo recursivo se hace patente al compararlo con otro no recursivo. Se emplaza al lector a que use su ingenio para escribir una solución no recursiva, que sea equivalente al programa anterior, antes de mirar la que se muestra en (4.41). Este programa se presenta sin más comentarios y puede servir de prueba para el lector descubrir cómo y por qué funciona.

```
program construirarbol(input, output);
type ref ↑nodo;
nodo record clave: integer;
    izquierdo, derecho: ref
end;
(4.41)
```

```
var i, n, ni, nd, x: integer;
raiz, q, r, s, ficticio: ref;
p: array [1 .. 30] of {pila}
record n: integer; rf: ref
end;
begin {el primer entero es el numero de nodos}
read(n); new(raiz); new(ficticio); {ficticio}
i := 1; p[1].n := n; p[1].rf := raiz;
repeat n := p[i].n; r := p[i].rf; i := i - 1; {quitar}
if n = 0 then r↑.derecho := nil else
begin s := ficticio;
repeat ni := n div 2; nd := n - ni - 1;
read(x); new(q); q↑.clave := x;
i := i + 1; p[i].n := nd; p[i].rf := q; {poner}
n := ni; s↑.izquierdo := q; s := q
until n = 0;
q↑.izquierdo := nil; r↑.derecho := ficticio↑.izquierdo
end
until i = 0;
imprimirarbol(raiz↑.derecho, 0)
end.
```

4.4.2. Operaciones básicas con árboles binarios

Puede haber muchas tareas a realizar con una estructura árbol; una muy corriente es ejecutar una determinada operación P con cada uno de los elementos del árbol. P se considera entonces como un parámetro de una tarea más general que es la visita de todos los nodos o, como se denomina usualmente, del recorrido del árbol.

Si se considera la tarea como un proceso secuencial, entonces los nodos individuales se visitan en un orden específico, y pueden considerarse como organizados según una estructura lineal. De hecho, se simplifica considerablemente la descripción de muchos algoritmos si puede hablarse del proceso del *siguiente* elemento en el árbol, según un cierto orden subyacente.

Hay tres órdenes de los elementos que están asociados de forma natural con la estructura de los árboles. Tal como sucede con la estructura en sí, éstos pueden expresarse adecuadamente en forma recursiva. Refiriéndose al árbol binario de la Fig. 4.24, donde R significa la raíz y A y B significan los subárboles izquierdo y derecho, las tres ordenaciones son:

1. *Pre-orden* : R, A, B (visitar la raíz *antes* que los subárboles)
2. *Orden Central*: A, R, B
3. *Post-orden* : A, B, R (visitar la raíz *después* que los subárboles)

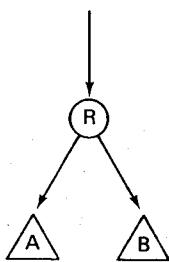


Fig. 4.24. Arbol binario.

Al recorrer el árbol de la Fig. 4.20 y escribir los caracteres de los nodos en la secuencia en que éstos se van encontrando, se obtienen las siguientes ordenaciones:

1. *Pre-orden* : $* + a / b c - d * e f$
2. *Orden Central*: $a + b / c * d - e * f$
3. *Post-orden* : $a b c / + d e f * - *$

Se pueden ver las tres formas de expresiones: el recorrido en pre-orden produce la notación *prefija*; el recorrido en post-orden genera la notación *postfija* o inversa y el recorrido en orden central produce la notación convencional, aunque sin los paréntesis necesarios para indicar la precedencia de los operadores.

A continuación se formulan los tres métodos de recorrido por medio de tres programas concretos, con el parámetro explícito *a* significando el árbol con el cual se opera, y el parámetro implícito *P* significando la operación a realizar con cada nodo. Se suponen las definiciones siguientes:

```

type ref = ↑nodo;
nodo = record . .
  izquierdo, derecho: ref
end
  
```

(4.42)

Los tres métodos pueden formularse ahora como procedimientos *recursivos*; una vez más se comprueba el hecho de que los algoritmos recursivos definen de la manera más conveniente las operaciones que trabajan con estructuras de datos definidas recursivamente.

```

procedure preorder(a: ref);
begin if a ≠ nil then
  begin P(a);
    preorder(a↑ .izquierdo);
    preorder(a↑ .derecho)
  end
end
  
```

(4.43)

```

procedure ordencentral(a: ref);
begin if a ≠ nil then
  begin ordencentral(a↑ .izquierdo);
    P(a);
    ordencentral(a↑ .derecho)
  end
end
  
```

(4.44)

```

procedure postorden(a: ref);
begin if a ≠ nil then
  begin postorden(a↑ .izquierdo);
    postorden(a↑ .derecho);
    P(a)
  end
end
  
```

(4.45)

Nótese que el puntero *a* es un parámetro constante (se pasa por valor). Esto expresa el hecho de que lo que interesa es la *referencia* al subárbol considerado y *no* la variable cuyo valor es el puntero, que podría ser alterada en caso de pasar *a* como parámetro variable (por referencia).

Un ejemplo de rutina que recorre un árbol es el procedimiento que imprime un árbol, indicando el nivel de cada nodo por medio de una jerarquización del margen apropiada (ver Programa 4.3).

Los árboles binarios se utilizan frecuentemente para representar conjuntos de datos cuyos elementos se identifican por una clave única. Si el árbol está organizado de tal manera que, para todo nodo *a_i*, todas las claves del subárbol izquierdo de *a_i* son menores que la clave de *a_i*, y todas aquellas en el subárbol derecho de *a_i* son mayores que la clave de *a_i*, se dice que este árbol es un *árbol de búsqueda*. Puede localizarse una clave arbitraria en un árbol de búsqueda empezando por la raíz, y avanzando por un camino de búsqueda de forma que la decisión de continuar por el subárbol izquierdo o derecho de un nodo dado se toma en base únicamente al valor de la clave de dicho nodo. Tal como se ha visto, un árbol binario de *n* elementos puede organizarse para que su altura no sea superior a $\log n$. Por lo tanto, si el árbol está perfectamente equilibrado, puede realizarse una búsqueda entre *n* elementos con no más de $\log n$ comparaciones. Obviamente el árbol es una forma de organizar conjuntos de datos de esta clase mucho más adecuada que la lista lineal utilizada en el apartado anterior.

Como esta búsqueda sigue un camino único desde la raíz hasta el nodo deseado, puede programarse fácilmente por medio de una iteración [(4.46)].

La función *loc(x, a)* toma el valor *nil* si no existe una clave con valor *x* en el árbol *a*. Tal como ocurrió en el caso de búsqueda en listas, la complejidad de la condición de terminación del bucle sugiere buscar una solución mejor. Esta consistía en el uso de un *centinela* al final de la lista. Puede aplicarse la misma idea al caso de un árbol. El uso de punteros permite que todas las ramas del árbol

```

function loc(x: integer; a: ref): ref;
  var encontrado: boolean;
begin encontrado := false;
  while(a ≠ nil) ∧ ¬encontrado do
    begin
      if a↑ .clave = x then encontrado := true else
        if a↑ .clave > x then a := a↑ .izquierdo else a := a↑ .derecho
    end;
    loc := a
  end

```

(4.46)

acaben con el mismo, idéntico, centinela. La estructura resultante no es ya un árbol, sino más bien un árbol con todas las hojas atadas a un punto de anclaje común, único (Fig. 4.25). El centinela puede ser considerado como un representante común de todos los nodos externos con los que se extendía el árbol original (ver Fig. 4.19). En (4.47) se presenta la rutina simplificada resultante.

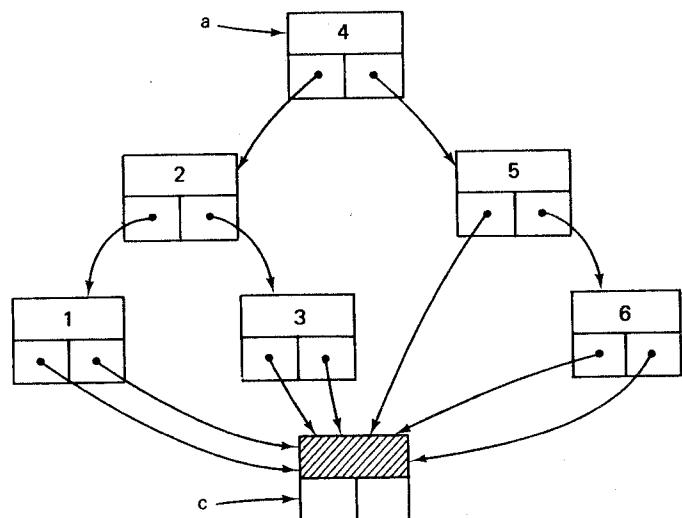


Fig. 4.25. Árbol de búsqueda con centinela.

```

function loc(x: integer; a: ref): ref;
begin c↑ .clave := x; {centinela}
  while a↑ .clave ≠ x do
    if x < a↑ .clave then a := a↑ .izquierdo else a := a↑ .derecho;
    loc := a
  end

```

(4.47)

Obsérvese que, en este caso, *loc(x, a)* toma el valor *c*, es decir el puntero al centi-

nela, si no se encuentra ninguna clave con valor *x* en el árbol de raíz *a*. *c* realiza el papel del puntero nil.

4.4.3. Búsqueda e inserción en árboles

Las aplicaciones en que se construye un conjunto de datos, y después éste no se modifica, no son buenos ejemplos para apreciar las capacidades de la técnica de asignación dinámica. Son ejemplos más apropiados aquellas aplicaciones en que la estructura del árbol varía, es decir, crece y/o disminuye durante la ejecución del programa. Estos son también los casos en que otras representaciones, tales como el array, fallan, y en las que el árbol con los elementos enlazados por punteros se presenta como la solución apropiada.

Se considerará primero el caso de un árbol que sólo crece, pero no disminuye nunca. Un ejemplo apropiado de esto es el problema de la concordancia que ya se estudió al tratar las listas enlazadas. Ahora se vuelve a examinar. En este problema se tiene una secuencia de palabras, y hay que determinar el número de veces que aparece cada una de ellas. Esto significa que, empezando con un árbol vacío, se busca cada palabra en el árbol. Si se encuentra, se incrementa su contador; si no, se inserta en el árbol como nueva palabra (con el contador inicializado a 1). Esta tarea se denomina *búsqueda en árbol con inserción*. Se suponen las siguientes definiciones de tipos de datos:

```

type ref = ↑palabra;
  palabra = record
    clave: integer;
    contador: integer;
    izquierdo, derecho: ref
  end

```

(4.48)

Suponiendo además un fichero de claves *f* y una variable que significa la raíz del árbol, se puede escribir el programa como

```

reset(f);
while ¬eof(f) do
  begin read(f, x); buscar(x, raiz) end

```

(4.49)

Encontrar el camino de búsqueda es, como antes, muy sencillo. Sin embargo, si éste lleva a un «callejón sin salida» (es decir, un subárbol vacío marcado por valor de puntero nil), la palabra que se procesa debe insertarse en el árbol, ocupando el sitio del subárbol vacío. Considérese, por ejemplo, el árbol binario de la Fig. 4.26 y la inserción de la palabra «Paul». La inserción resultante aparece con líneas de trazos en la misma figura.

La operación completa se muestra en el Programa 4.4. El proceso de búsqueda se formula como procedimiento recursivo. Obsérvese que el parámetro *p* es un

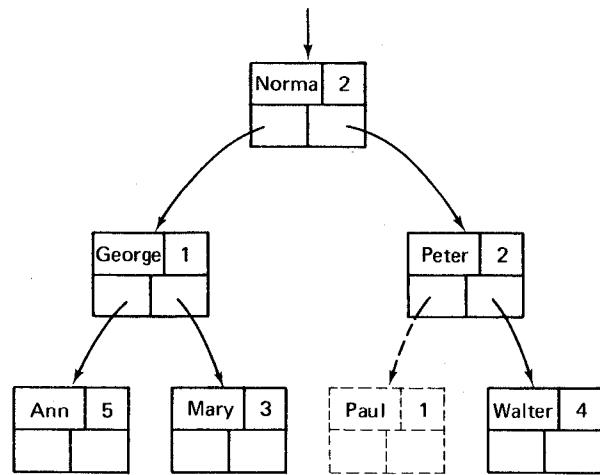


Fig. 4.26. Inserción en un árbol binario ordenado.

parámetro variable y *no* uno constante. Esto es esencial porque cuando se presenta el caso de inserción se debe asignar un nuevo valor a la *variable* que contenía el valor **nil**. Utilizando, como secuencia de entrada, los mismos 21 números que se emplearon en el Programa 4.3 para construir el árbol de la Fig. 4.23, el Programa 4.4 produce el árbol binario de búsqueda que aparece en la Fig. 4.27.

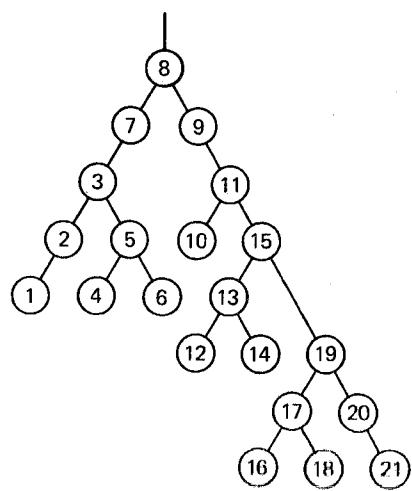


Fig. 4.27. Arbol de búsqueda generado por el Programa 4.4.

Programa 4.4. Búsqueda e inserción en árbol.

```

program busquedaaenarbol(input, output);
{busqueda e insercion en arbol binario}
type ref = ↑palabra;
palabra = record clave: integer;
contador: integer;
izquierdo, derecho: ref
end;
var raiz: ref; k: integer;
procedure imprimirarbol(p: ref; n: integer);
var i: integer;
begin if p ≠ nil then
with p↑ do
begin imprimirarbol(izquierdo, n + 1);
for i := 1 to n do write("      ");
writeln(clave);
imprimirarbol(derecho, n + 1)
end
end;
procedure buscar(x: integer; var p: ref);
begin
if p = nil then
begin {la palabra no esta en el arbol; insertarla}
new(p);
with p↑ do
begin clave := x; contador := 1; izquierdo := nil; derecho := nil
end
end else
if x < p↑ .clave then buscar(x, p↑ .izquierdo) else
if x > p↑ .clave then buscar(x, p↑ .derecho) else
p↑ .contador := p↑ .contador + 1
end {buscar};
begin raiz := nil;
while ¬eof(input) do
begin read(k); buscar(k, raiz)
end;
imprimirarbol(raiz, 0)
end .
  
```

Nuevamente el uso de un centinela simplifica algo el trabajo, tal como se indica en (4.50). Está claro que ahora la variable *raíz* debe ser inicializada al principio

del programa con el puntero al nodo centinela, en vez de con el valor **nil**, y, antes de cada búsqueda, el valor buscado x debe asignarse al campo de la clave del centinela.

```

procedure buscar(x: integer; var p: ref);
begin
  if x < p↑ .clave then buscar(x, p↑ .izquierdo) else
  if x > p↑ .clave then buscar(x, p↑ .derecho) else
  if p ≠ c then p↑ .contador := p↑ .contador + 1 else
    begin {insertar} new(p);
    with p↑ do
      begin clave := x; izquierdo := c; derecho := c; contador := 1
      end
    end
  end
end

```

(4.50)

Una vez más, la última, se va a desarrollar una versión alternativa de este programa, esta vez sin utilizar la recursión. Evitar la recursión no es tan trivial ahora como cuando no hay que hacer una inserción, pues, si la hay, debe recordarse, al menos, el último paso del camino recorrido. Esto se ha conseguido automáticamente en el Programa 4.4 haciendo uso de un parámetro variable.

```

procedure buscar(x: integer; raiz: ref);
  var p1, p2: ref; d: integer;
begin p2 := raiz; p1 := p2↑ .derecho; d := 1;
  while (p1 ≠ nil) ∧ (d ≠ 0) do
    begin p2 := p1;
    if x < p1↑ .clave then
      begin p1 := p1 .izquierdo; d := -1 end else
    if x > p1↑ .clave then
      begin p1 := p1 .derecho; d := 1 end else
    d := 0
    end;
  if d = 0 then p1↑ .contador := p1 .contador + 1 else
    begin {insertar} new(p1);
    with p1↑ do
      begin clave := x; izquierdo := nil; derecho := nil; contador := 1
      end;
    if d < 0 then p2 .izquierdo := p1 else p2 .derecho := p1
    end
  end
end

```

(4.51)

Para enlazar correctamente el componente insertado, hay que conocer la referencia a su antecesor y saber si tiene que insertarse como subárbol izquierdo o como subárbol derecho. A tal fin, se introducen dos variables llamadas $p2$ y d (de dirección).

Como en el caso de búsqueda e inserción en lista, se utilizan dos punteros $p1$ y $p2$, que recorren el camino de búsqueda de manera que $p2$ apunta siempre al antecesor de $p1$. Para iniciar el proceso con esta condición satisfecha, se introduce un elemento auxiliar, ficticio, referenciado por el puntero llamado *raiz*. El principio del verdadero árbol de búsqueda está referenciado por el puntero $raiz$ [↑] .derecho. Por lo tanto, el programa debe comenzar con las instrucciones

new(raiz); raiz[↑] .derecho := nil

en vez de con la asignación

raiz := nil

Aunque el objeto de este algoritmo es insertar, puede utilizarse también para ordenar. De hecho, se asemeja mucho al método de ordenación por inserción, y, al utilizar una estructura árbol en lugar de un array, desaparece la necesidad de reubicar los componentes que se encuentran por encima del punto de inserción. La ordenación con árboles puede programarse para que sea casi tan eficiente como los mejores métodos conocidos de ordenación con arrays. Pero deben tomarse algunas precauciones. Desde luego, ahora hay que tratar de forma diferente el caso en que se encuentra una clave coincidente con otra ya insertada. Si el caso $x = p1$.clave se trata de forma idéntica al caso $x > p1$.clave, entonces el algoritmo es un método de ordenación estable, es decir, los elementos con claves idénticas aparecen, cuando se examina el árbol en el orden normal, en la misma secuencia que cuando fueron insertados.

En general hay métodos de ordenación mejores pero, en aplicaciones donde se necesita a la vez buscar y ordenar, el algoritmo de búsqueda e inserción en árbol es muy recomendable. De hecho, se aplica frecuentemente en compiladores y bancos de datos para organizar los objetos que van a ser almacenados y luego recuperados. Un ejemplo apropiado lo constituye la construcción de un *índice de referencias cruzadas* de un texto dado. A continuación se estudia este problema en detalle.

Hay que construir un programa que, mientras lee un texto f y lo imprime incluyendo números de línea consecutivos, guarde todas las palabras de este texto, junto con los números de las líneas en las que cada palabra aparece. Cuando se acaba de procesar el texto, hay que imprimir una tabla con todas las palabras aparecidas en el texto, en orden alfabético, junto con listas de los números de línea donde aparecen.

Resulta obvio que el árbol de búsqueda (también llamado árbol *lexicográfico*) es un candidato ideal para representar las palabras que aparecen en el texto.

Ahora cada nodo no sólo contiene una palabra como valor de la clave, sino que también es cabecera de una lista de números de línea. Se llamará *ítem* a cada registro de aparición de una palabra. Por lo tanto, en este ejemplo se presentan árboles y listas lineales al mismo tiempo. El programa está formado por dos partes principales (ver Programa 4.5), la fase donde se examina el texto y la fase en que se imprime la tabla. Esta última es una aplicación directa de la rutina de recorrido de un árbol, donde visitar cada nodo implica imprimir el valor de la clave (palabra) y el examen de su lista asociada de números de línea (ítems). A continuación se dan más aclaraciones sobre el *Generador de Referencias Cruzadas*, Programa 4.5:

1. Se considera como palabra una secuencia de letras y dígitos que comience con una letra.
2. La clave se forma con los $c1$ primeros caracteres solamente. Por lo tanto, se consideran idénticas dos palabras cuyos $c1$ primeros caracteres sean iguales.
3. Los $c1$ caracteres se empaquetan en un array *id* (tipo *alfa*). Si $c1$ es suficientemente pequeño, muchos computadores podrán comparar tales arrays empaquetados con una única instrucción.
4. La variable *k1* se utiliza como el índice que mantiene la siguiente condición invariante del buffer de caracteres *a*:

$$a[i] = '' \quad \text{para } i = k1 + 1 \dots c1$$

Las palabras que tienen menos de $c1$ caracteres son ampliadas con un número apropiado de espacios en blanco.

5. Es conveniente que los números de línea se impriman en orden ascendente. Por lo tanto, las listas de ítems deben ser generadas en el mismo orden en que van a ser examinadas al imprimir. Esto sugiere el uso de dos punteros en cada nodo palabra, uno apuntando al primer ítem de la lista, y otro apuntando al último.
6. La fase que examina el texto está construida de tal forma que se omiten del índice las palabras que estén entre comillas o dentro de comentarios, suponiendo que los textos entre comillas y comentarios no se extienden más allá de los finales de línea.

Programa 4.5. Generador de referencias cruzadas.

```
program refcruzadas(f, output);
{generador de referencias cruzadas utilizando un arbol binario}
const c1 = 10;    {longitud de las palabras}
c2 = 8;          {numeros por linea}
c3 = 6;          {digitos por numero}
c4 = 9999;        {numero de linea maximo}
```

```
type alfa = packed array [1 .. c1] of char;
respalabra = ↑palabra;
refitem = ↑item;
palabra = record clave: alfa;
            primero, ultimo: refitem;
            izquierdo, derecho: respalabra
          end;
item = packed record
            nol: 0 .. c4;
            sig: refitem
          end;
var raiz: respalabra;
k, k1: integer;
n: integer;           {numero de linea en curso}
id: alfa;
f: text;
a: array [1 .. c1] of char;
procedure buscar (var p1: respalabra);
var p: respalabra; x: refitem;
begin p := p1;
if p = nil then
begin new(p); new(x);
with p↑ do
begin clave := id; izquierdo := nil;
derecho := nil; primero := x; ultimo := x
end;
x↑ .nol := n; x↑ .sig := nil; p1 := p;
end else
if id < p↑ .clave then buscar(p↑ .izquierdo) else
if id > p↑ .clave then buscar(p↑ .derecho) else
begin new(x); x↑ .nol := n; x↑ .sig := nil;
p↑ .ultimo↑ .sig := x; p↑ .ultimo := x
end
end {buscar};
procedure imprimirarbol(p: respalabra);
procedure imprimirpalabra(p: palabra);
var l: integer; x: refitem;
begin write(' ', p .clave);
x := p .primero; l := 0;
repeat if l = c2 then
begin writeln;
l := 0; write(' ': c1 + 1)
end;
```

Programa 4.5. (Continuación)

```

 $l := l + 1; write(x \uparrow .nol: c3); x := x \uparrow .sig$ 
until  $x = \text{nil}$ ;
writeln
end {imprimirpalabra};
begin if  $p \neq \text{nil}$  then
  begin imprimirarbol( $p \uparrow .izquierdo$ );
    imprimirpalabra( $p \uparrow$ ); imprimirarbol( $p \uparrow .derecho$ )
  end
end {imprimirarbol};
begin raiz := nil; n := 0; k1 := c1;
page(output); reset(f);
while  $\neg eof(f)$  do
begin if  $n = c4$  then  $n := 0$ ;
   $n := n + 1; write(n: c3); \{linea siguiente\}$ 
  write(' ');
  while  $\neg eoln(f)$  do
    begin {examinar linea}
      if  $f \uparrow$  in ['A' .. 'Z'] then
        begin k := 0;
          repeat if  $k < c1$  then
            begin k := k + 1; a[k] := f \uparrow;
            end;
            write(f \uparrow); get(f)
          until  $\neg (f \uparrow$  in ['A' .. 'Z', '0' .. '9']);
          if  $k \geq k1$  then  $k1 := k$  else
            repeat a[k1] := ' '; k1 := k1 - 1
            until  $k1 = k$ ;
          pack(a, 1, id); buscar(raiz)
        end else
        begin {ver si es comilla o comentario}
          if  $f \uparrow = " "$  then
            repeat write(f \uparrow); get(f)
            until  $f \uparrow = " "$  else
          if  $f \uparrow = '{$  then
            repeat write(f \uparrow); get(f)
            until  $f \uparrow = '}'$ ;
            write(f \uparrow); get(f)
          end
        end;
        writeln; get(f)
      end
      page(output); imprimirarbol(raiz)
    end.

```

Programa 4.5. (Continuación)

La Tabla 4.4 muestra el resultado de procesar un pequeño texto de programa.

Tabla 4.4. Resultado típico del Programa 4.5.

```

1 PROGRAM PERMUTAR (OUTPUT);
2 CONST N = 4;
3 VAR I: INTEGER;
4 A: ARRAY [1..N] OF INTEGER;
5
6 PROCEDURE IMPRIMIR;
7   VAR I: INTEGER;
8   BEGIN FOR I := 1 TO N DO WRITE (A[I]:3);
9     WRITELN
10  END {IMPRIMIR} ;
11
12 PROCEDURE PERM (K: INTEGER);
13   VAR I,X: INTEGER;
14   BEGIN
15     IF K = 1 THEN IMPRIMIR ELSE
16       BEGIN PERM (K-1);
17         FOR I := 1 TO K-1 DO
18           BEGIN X := A[I]; A[I] := A[K]; A[K] := X;
19             PERM (K-1);
20             X := A[I]; A[I] := A[K]; A[K] := X;
21           END
22         END
23  END {PERM} ;
24
25 BEGIN
26   FOR I := 1 TO N DO A[I] := I;
27   PERM (N)
28 END .

```

	ARRAY	4	4	8	18	18	18	18	20	20
A			20	20	26					
BEGIN		8	14	16	18	25				
CONST			2							
DO		8	17	26						
ELSE			15							
END		10	21	22	23	28				
FOR		8	17	26						
IF			15							

IMPRIMIR	6	15						
ENTERO	3	4	7	12	13			
I	3	7	8	8	13	17	18	18
	20	20	26	26	26			
K	12	15	16	17	18	18	19	20
	20							
N	2	4	8	26	27			
OF	4							
OUTPUT	1							
PERMUTAR	1							
PERM	12	16	19	27				
PROCEDURE	6	12						
PROGRAM	1							
THEN	15							
TO	8	17	26					
VAR	3	7	13					
WRITELN	9							
WRITE	8							
X	13	18	18	20	20			

Tabla 4.4. (Continuación)

4.4.4. Borrado en árboles

A continuación se estudia el problema inverso de la inserción, el borrado. La tarea consiste en borrar, es decir, sacar el nodo con clave x de un árbol que tiene las claves ordenadas. Desgraciadamente, sacar un elemento no es en general tan simple como insertarlo. Es una tarea fácil si el elemento a borrar es un nodo terminal o tiene un único descendiente. La dificultad está en el borrado de un elemento que tiene dos descendientes, pues no puede señalarse en dos direcciones con un solo puntero. En esta situación, el elemento a borrar debe ser reemplazado, bien por el elemento más a la derecha en el subárbol izquierdo, o bien por el nodo más a la izquierda en el subárbol derecho, los dos como máximo con un único descendiente. Los detalles del proceso se muestran en el procedimiento recursivo llamado *borrar* (4.52). Este procedimiento distingue tres casos:

1. No hay ningún componente con clave igual a x .
2. El componente con clave x tiene un único descendiente como máximo.
3. El componente con clave x tiene dos descendientes.

El procedimiento auxiliar recursivo *bor* se activa sólo en el caso 3. Entonces «desciende» a lo largo de la rama más a la derecha del subárbol izquierdo del elemento que se va a borrar q^\uparrow , y reemplaza la información de interés en q^\uparrow por los correspondientes valores del componente, d^\uparrow , más a la derecha en ese subárbol izquierdo. El procedimiento, no especificado, *dispose(q)*, puede considerarse

```

procedure borrar (x: integer; var p: ref);
  var q: ref;
  procedure bor (var d: ref);
  begin if d^.derecho ≠ nil then bor(d^.derecho) else
    begin q^.clave := d^.clave; q^.contador := d^.contador;
      q := d; d := d^.izquierdo
    end
  end;
begin {borrar}
  if p = nil then writeln ('LA PALABRA NO ESTA EN EL ARBOL') else
    if x < p^.clave then borrar (x, p^.izquierdo) else
      if x > p^.clave then borrar(x, p^.derecho) else
        begin {borrar p↑} q := p;
          if q^.derecho = nil then p := q^.izquierdo else
            if q^.izquierdo = nil then p := q^.derecho else
              bor (q^.izquierdo) {dispose(q)}
        end
  end {borrar}
end {borrar}

```

(4.52)

como el inverso u opuesto de *new(q)*. El segundo asigna memoria para un nuevo componente, pero él primero puede servir para indicar a un sistema computador que la memoria utilizada por q^\uparrow queda otra vez disponible y puede ser reutilizada (se «recicla» la memoria).

En la Fig. 4.28 se indica cómo funciona el procedimiento (4.52). Considérese el árbol (a); sacando sucesivamente los nodos con claves 13, 15, 5, 10, se obtienen los árboles de las Fig. 4.28 (b-e).

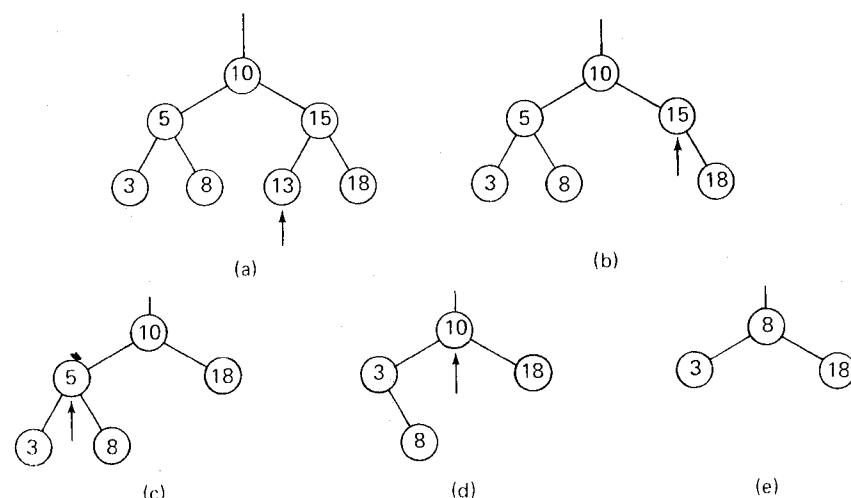


Fig. 4.28. Borrado en árbol.

4.4.5. Análisis de la búsqueda e inserción en árboles

Es una reacción natural —y sana— desconfiar del algoritmo de búsqueda e inserción en árbol. Al menos, se debe tener cierto escepticismo hasta conocer algunos detalles adicionales sobre su comportamiento. Lo que preocupa a muchos programadores al principio es que no se sabe cómo va a crecer el árbol; no se tiene idea de la forma que va a tomar. Sólo puede adivinarse que, muy probablemente, no será un árbol perfectamente equilibrado. Como el número de comparaciones necesarias para encontrar una clave en un árbol perfectamente equilibrado de n nodos es aproximadamente $h = \log n$, el número de comparaciones en un árbol generado por este algoritmo será mayor que h . Pero, la pregunta surge, ¿en qué medida mayor?

Antes que nada, es fácil encontrar el caso más desfavorable. Supóngase que todas las claves llegan ya en orden estrictamente ascendente (o descendente). Entonces cada clave se coloca inmediatamente a la derecha (izquierda) de su predecesor, y el árbol resultante degenera completamente, es decir, se convierte en una lista lineal. El trabajo necesario por término medio resulta ser, en este caso, $n/2$ comparaciones. Evidentemente, este caso más desfavorable conduce a un resultado del algoritmo de búsqueda muy pobre, y parece justificar plenamente el escepticismo anterior. Falta saber, desde luego, con qué frecuencia se presentará este caso. Dicho de forma más precisa, sería bueno conocer la longitud a_n del camino de búsqueda, promediada entre todas las n claves, y entre todos los $n!$ árboles que pueden ser generados a partir de las $n!$ permutaciones de las n claves distintas originales. Este problema de análisis algorítmico resulta ser bastante sencillo, y se presenta aquí tanto como ejemplo típico de análisis de un algoritmo como por la importancia práctica de su resultado.

Se suponen n claves distintas dadas, con valores $1, 2, \dots, n$. Supóngase que llegan en un orden aleatorio. La probabilidad de que la primera clave que aparezca

que resulta ser el nodo raíz —tenga el valor i es $1/n$. Eventualmente, el subárbol izquierdo contendrá $i - 1$ nodos, y el subárbol derecho $n - i$ nodos (ver Figura 4.29).

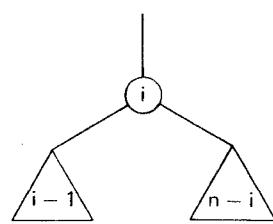


Fig. 4.29. Distribución de pesos en las ramas.

Sea a_{i-1} el camino medio en el subárbol izquierdo, y a_{n-i} el del subárbol derecho, suponiendo de nuevo que todas las permutaciones posibles de las $n - 1$ claves restantes son igualmente probables. La longitud de camino media en un árbol con n nodos es la suma de los productos del nivel de cada nodo multi-

plicado por su probabilidad de acceso. Si ésta se supone la misma para todos ellos se tiene

$$a_n = \frac{1}{n} \sum_{i=1}^n c_i \quad (4.53)$$

siendo c_i la longitud de camino del nodo i .

Se dividen los nodos del árbol de la Fig. 4.29 en tres clases:

1. Los $i - 1$ nodos del subárbol izquierdo tienen una longitud de camino media igual a $a_{i-1} + 1$.
2. La raíz tiene una longitud de camino igual a 1.
3. Los $n - i$ nodos del subárbol derecho tiene una longitud de camino media igual a $a_{n-i} + 1$.

Por lo tanto, (4.53) puede expresarse como una suma de tres términos.

$$a_n^{(i)} = (a_{i-1} + 1) \frac{i-1}{n} + 1 \cdot \frac{1}{n} + (a_{n-i} + 1) \frac{n-i}{n} \quad (4.54)$$

La cantidad deseada a_n se obtiene ahora promediando $a_n^{(i)}$ para todos los $i = 1 \dots n$, es decir, para todos los árboles con clave 1, 2, ..., n en la raíz.

$$\begin{aligned} a_n &= \frac{1}{n} \sum_{i=1}^n \left[(a_{i-1} + 1) \frac{i-1}{n} + \frac{1}{n} + (a_{n-i} + 1) \frac{n-i}{n} \right] \\ &= 1 + \frac{1}{n^2} \sum_{i=1}^n [(i-1)a_{i-1} + (n-i)a_{n-i}] \\ &= 1 + \frac{2}{n^2} \sum_{i=1}^n (i-1)a_{i-1} = 1 + \frac{2}{n^2} \sum_{i=1}^{n-1} i \cdot a_i \end{aligned} \quad (4.55)$$

La ecuación (4.55) es una relación de recurrencia en a_n de la forma $a_n = f_1(a_1, a_2, \dots, a_{n-1})$. A partir de ella puede obtenerse una relación de recurrencia más simple del tipo $a_n = f_2(a_{n-1})$, como sigue:

De (4.45) resulta directamente

$$(1) \quad a_n = 1 + \frac{2}{n^2} \sum_{i=1}^{n-1} i \cdot a_i = 1 + \frac{2}{n^2} (n-1)a_{n-1} + \frac{2}{n^2} \sum_{i=1}^{n-2} i \cdot a_i$$

$$(2) \quad a_{n-1} = 1 + \frac{2}{(n-1)^2} \sum_{i=1}^{n-2} i \cdot a_i$$

Multiplicando (2) por $((n-1)/n)^2$, se obtiene

$$(3) \quad \frac{2}{n^2} \sum_{i=1}^{n-2} i \cdot a_i = \frac{(n-1)^2}{n^2} (a_{n-1} - 1)$$

y sustituyendo (3) en (1) se obtiene

$$a_n = \frac{1}{n^2}((n^2 - 1)a_{n-1} + 2n - 1) \quad (4.56)$$

Sucede que a_n puede expresarse de forma no recursiva, cerrada, en términos de la función armónica

$$\begin{aligned} H_n &= 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \\ a_n &= 2\frac{n+1}{n}H_n - 3 \end{aligned} \quad (4.57)$$

[El lector escéptico debe comprobar que (4.57) satisface la relación recurrente (4.56).]

A partir de la fórmula de Euler (utilizando la constante de Euler $\gamma \approx 0.577$)

$$H_n = \gamma + \ln(n) + \frac{1}{12n^2} + \cdots$$

se deduce, para n grande, la relación

$$a_n \approx 2[\ln(n) + \gamma] - 3 = 2\ln(n) - c$$

Como la longitud de camino media del árbol perfectamente equilibrado es aproximadamente

$$a'_n = \log(n) - 1 \quad (4.58)$$

se obtiene, despreciando los términos constantes que, para n grande, se vuelven insignificantes:

$$\lim_{n \rightarrow \infty} \frac{a_n}{a'_n} = \frac{2 \ln n}{\log n} = 2 \cdot \ln 2 \approx 1.386 \quad (4.59)$$

¿Qué significa el resultado (4.59)? Significa que, tomándose el trabajo de construir siempre un árbol perfectamente equilibrado, en vez del árbol «aleatorio» obtenido con el Programa 4.4, se podría —siempre suponiendo que todas las claves se buscan con la misma probabilidad— esperar una mejora en la longitud de camino media de un 39 % como mucho. Hay que enfatizar la palabra «media» ya que, desde luego, la mejora puede ser mucho mayor en el caso desafortunado de que el árbol construido hubiera degenerado completamente en una lista, caso que, sin embargo, es muy improbable que ocurra (si todas las permutaciones de las n claves a insertar son igualmente probables). A este respecto hay que hacer notar que la longitud de camino media del árbol «aleatorio» aumenta también

con el número de nodos de una forma estrictamente logarítmica, incluso aunque en el caso más desfavorable la longitud de camino crezca linealmente.

La cantidad de 39 % marca un límite en el esfuerzo adicional que puede hacerse, de manera eficaz, en cualquier tipo de reorganización del árbol, cuando se insertan las claves. Naturalmente, el cociente q , entre las frecuencias de acceso (recuperación) de nodos (información) y de inserción de los mismos, afecta significativamente al umbral de beneficio obtenido con este trabajo. Cuanto mayor es este cociente, mayor es el rendimiento del procedimiento reorganizador. La cantidad de 39 % es lo suficientemente baja como para que, en la mayor parte de las aplicaciones, no sea rentable mejorar el algoritmo de inserción simple a menos que el número de nodos y el cociente accesos/inserciones sean suficientemente grandes (o si se teme que se pueda presentar el caso más desfavorable).

4.4.6. Árboles equilibrados

A partir de las consideraciones anteriores, está claro que un procedimiento de inserción que siempre restaure la estructura del árbol a un equilibrio perfecto tiene muy pocas posibilidades de ser eficaz, ya que la restauración del equilibrio perfecto, después de realizar una inserción al azar, es una operación bastante compleja. Las posibles mejoras están en una formulación menos estricta del «equilibrio». Tales criterios de equilibrio «imperfecto» deberían conducir a procedimientos de reorganización más simples, a costa de un pequeño deterioro en el rendimiento medio de la búsqueda.

Una definición de equilibrio de esta clase ha sido postulada por Adelson-Velskii y Landis [4.1]. El criterio de equilibrio es el siguiente:

Un árbol está *equilibrado* si, y sólo si, para cada uno de sus nodos ocurre que las alturas de sus dos subárboles difieren como mucho en 1.

Los árboles que cumplen esta condición son a menudo llamados árboles AVL (en honor de sus inventores). Aquí se llamarán *árboles equilibrados* simplemente, ya que este criterio de equilibrio parece ser uno de los más adecuados. (Obsérvese que todos los árboles perfectamente equilibrados son también equilibrados en el sentido AVL.)

La definición no sólo es simple, sino que además conduce a un procedimiento de reequilibrado relativamente sencillo, y a una longitud de camino media prácticamente idéntica a la del árbol perfectamente equilibrado.

En un árbol equilibrado, se pueden realizar en $O(\log n)$ unidades de tiempo, incluso en el peor de los casos, las siguientes operaciones:

1. Encontrar un nodo con una clave dada.
2. Insertar un nodo con una clave dada.
3. Borrar un nodo con una clave dada.

Estos resultados son consecuencias directas de un teorema demostrado por Adelson-Velskii y Landis, según el cual un árbol equilibrado nunca será más

de un 45 % más alto que su correspondiente perfectamente equilibrado, independientemente del número de nodos que tenga.

Si se designa la altura de un árbol equilibrado de n nodos por $h_e(n)$, entonces

$$\log(n+1) \leq h_e(n) \leq 1.4404 \cdot \log(n+2) - 0.328 \quad (4.60)$$

Desde luego, se consigue el óptimo si el árbol está perfectamente equilibrado para $n = 2^k - 1$. Pero, ¿cuál es la estructura del peor árbol AVL?

Para encontrar la máxima altura h de todos los árboles equilibrados de n nodos, considérese primero una h fija e intétense construir el árbol equilibrado que tenga el mínimo número de nodos. Se recomienda esta estrategia porque, lo mismo que en el caso de la h mínima, el valor de ésta puede obtenerse sólo para determinados valores específicos de n . Se designa este árbol de altura h por T_h . Está claro que T_0 es el árbol vacío y T_1 es el árbol con un único nodo. Para construir el árbol T_h con $h > 1$, se pondrá una raíz con dos subárboles que a su vez tengan el menor número de nodos. Por lo tanto, los subárboles son también del tipo T . Evidentemente, uno de los subárboles *debe* tener altura $h - 1$, y el otro puede entonces tener una altura de una unidad menos, es decir,

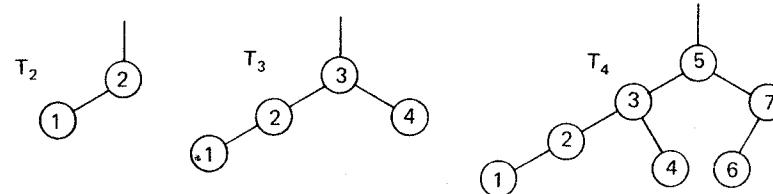


Fig. 4.30. Árboles Fibonacci de alturas 2, 3 y 4.

$h = 2$. La Fig. 4.30 muestran los árboles con alturas 2, 3 y 4. Como su mecanismo de formación se parece mucho al de los números de Fibonacci, se les llama *árboles Fibonacci*. Se definen como:

1. El árbol vacío es el árbol Fibonacci de altura 0.
2. Un nodo único es el árbol Fibonacci de altura 1.
3. Si T_{h-1} y T_{h-2} son árboles Fibonacci de alturas $h - 1$ y $h - 2$, entonces $T_h = \langle T_{h-1}, x, T_{h-2} \rangle$ es el árbol Fibonacci de altura h .
4. No hay otros árboles que sean árboles Fibonacci.

El número de nodos de T_h viene dado por la sencilla relación recurrente:

$$\begin{aligned} N_0 &= 0, & N_1 &= 1 \\ N_h &= N_{h-1} + 1 + N_{h-2} \end{aligned} \quad (4.61)$$

Donde los N_i son los números para los que se presenta el caso más desfavorable de (4.60) (límite superior de h).

4.4.7. Inserción en árboles equilibrados

Considérese ahora lo que puede ocurrir al insertar un nuevo nodo en un árbol equilibrado. Dada una raíz r con subárboles izquierdo y derecho I y D , hay que distinguir tres casos. Supóngase que el nuevo nodo se inserta en I haciendo que su altura aumente en 1:

1. $h_I = h_D$: I y D tendrán distinta altura, pero el criterio de equilibrio no se vulnera.
2. $h_I < h_D$: I y D igualarán sus alturas, es decir, incluso se mejora el equilibrio.
3. $h_I > h_D$: Se vulnera el criterio de equilibrio y hay que reestructurar el árbol.

Considérese el árbol de la Fig. 4.31. Los nodos con claves 9 y 11 pueden ser insertados sin necesidad de reequilibrar el árbol; el árbol de raíz 10 tendrá un solo subárbol —bien el izquierdo o el derecho— (caso 1), y el árbol con raíz 8 mejorará su equilibrio (caso 2). Sin embargo, la inserción de uno cualquiera de los nodos con claves 1, 3, 5 o 7, exigirá un reequilibrado posterior.

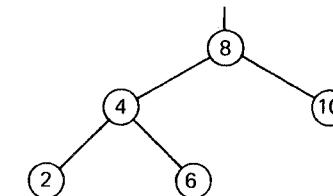


Fig. 4.31. Árbol equilibrado.

Un análisis cuidadoso de la situación revela que solo hay dos organizaciones del árbol que tienen que considerarse individualmente. El resto de ellas puede deducirse por simetría a partir de esas dos. El caso 1 se presenta al insertar las claves 1 ó 3 en el árbol de la Fig. 4.31, y el caso 2 al insertar los nodos 5 ó 7.

Los dos casos se generalizan en la Fig. 4.32 donde las cajas rectangulares representan subárboles, y la altura añadida por la inserción se indica con cruces. Simples transformaciones de las dos estructuras restauran el equilibrio deseado.

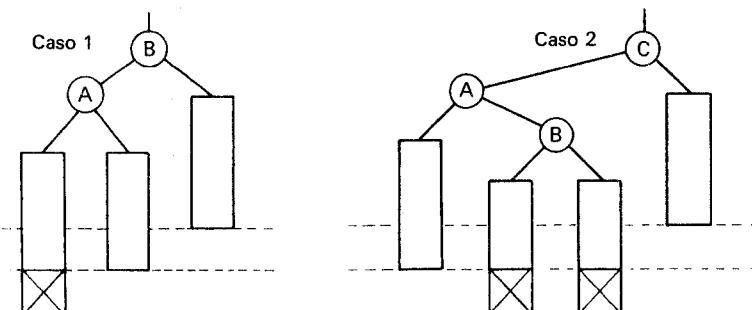


Fig. 4.32. Desequilibrio producido por la inserción.

El resultado se muestra en la Fig. 4.33; obsérvese que los únicos movimientos permitidos son los que se realizan en dirección vertical, mientras que las posiciones horizontales relativas de los nodos y subárboles mostrados deben permanecer invariables.

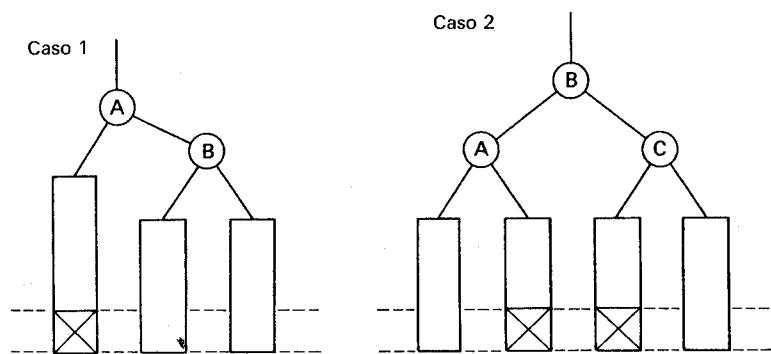


Fig. 4.33. Restauración del equilibrio.

Un algoritmo que inserte y reequilibre dependerá en forma crítica de la forma en que se almacene la información relativa al equilibrio del árbol. Una solución extrema consiste en mantener la información sobre el equilibrio, de forma completamente implícita, en la estructura misma del árbol. En este caso, sin embargo, el factor de equilibrio de cada nodo debe «averiguarlo» cada vez que se encuentre afectado por una inserción, con el consiguiente trabajo resultante. La otra solución extrema consiste en atribuir a, y almacenar con, cada nodo un factor de equilibrio explícito. La definición de un nodo (4.48) se amplía entonces a

```
type nodo = record clave: integer;
            contador: integer;
            izquierdo, derecho: ref;
            equi: -1 .. +1
          end
```

(4.62)

El factor de equilibrio de un nodo se interpretará subsiguientemente como la altura del subárbol derecho menos la altura del subárbol izquierdo, y se basará el algoritmo resultante en la definición de nodo dada en (4.62).

El proceso de inserción de un nodo consta fundamentalmente de las tres partes consecutivas siguientes:

1. Seguir el camino de búsqueda hasta que se comprueba que la clave aún no está en el árbol.
2. Insertar el nuevo nodo y determinar el factor de equilibrio resultante.
3. Volver siguiendo el camino de búsqueda y comprobar el factor de equilibrio de cada nodo.

Aunque este método realiza algunas comprobaciones que son redundantes (una vez que se establece el equilibrio, éste no necesita comprobarse en los antecesores del nodo en cuestión), se trabajará primero con este método, que es evidentemente correcto, ya que puede programarse con solo extender el procedimiento de búsqueda e inserción, ya descrito, del Programa 4.4. Este algoritmo describe la operación de búsqueda requerida en cada nodo individual y, debido a su formulación recursiva, puede fácilmente albergar una operación adicional «en el camino de vuelta atrás a lo largo del camino de búsqueda». En cada paso, se debe mandar información sobre si la altura del subárbol (en el cual se ha realizado la inserción) ha aumentado o no. Por lo tanto, se extiende la lista de parámetros del procedimiento con el boolean *h* que significa «la altura del subárbol ha aumentado». Está claro que *h* debe ser un parámetro de tipo variable, ya que se utiliza para transmitir un resultado.

Supóngase ahora que el proceso vuelve al nodo $p \uparrow$ desde la rama izquierda (ver Fig. 4.32), con la indicación de que su altura ha aumentado. Hay que distinguir tres situaciones, según las alturas de los subárboles previas a la inserción:

1. $h_I < h_D, p \uparrow .equi = +1$, el desequilibrio anterior ha sido corregido.
2. $h_I = h_D, p \uparrow .equi = 0$, el peso está ahora desplazado hacia la izquierda.
3. $h_I > h_D, p \uparrow .equi = -1$, se debe reequilibrar.

En el tercero de los casos, una inspección del factor de equilibrio del subárbol izquierdo (es decir, $p1 \uparrow .equi$) determina si se trata del caso 1 ó del caso 2 de la Fig. 4.32. Si ese nodo tiene también un subárbol izquierdo mayor que el derecho, entonces se trata del caso 1, sino del caso 2. (Convéñase el lector de que no se puede tener en este caso un subárbol izquierdo con un factor de equilibrio igual a 0 en su raíz.) Las operaciones necesarias para reestablecer el equilibrio se expresan en su totalidad como secuencias de reasignaciones de punteros. De hecho, los punteros se intercambian cíclicamente, resultando una rotación simple o doble de los dos o tres nodos en cuestión. Además de la rotación de los punteros, se deben también ajustar los respectivos factores de equilibrio de los nodos. Los detalles del proceso se presentan en el procedimiento de búsqueda, inserción y reequilibrado (4.63).

El mecanismo de trabajo se muestra en la Fig. 4.34. Considérese el árbol binario (a) que está formado por dos únicos nodos. La inserción de la clave 7 produce primero un árbol desequilibrado (una lista lineal). Se equilibra con una rotación *DD* simple, resultando el árbol perfectamente equilibrado (b). A continuación, una inserción de los nodos 2 y 1 produce un desequilibrio del subárbol con raíz 4. Este subárbol se equilibra con una rotación *II* simple (d). La inserción subsiguiente de la clave 3 anula inmediatamente el equilibrio del nodo raíz 5. El equilibrio se restaura a continuación mediante la rotación *ID* doble, que es más complicada; el resultado es el árbol (e). El único nodo que puede ahora perder el equilibrio con una inserción es el 5. De hecho, la inserción del nodo 6 debe

```

procedure buscar (x: integer; var p: ref; var h: boolean);
  var p1, p2: ref; {h = falso}
begin
  if p = nil then
    begin {la palabra no esta en el arbol; insertarla}
      new(p); h := true;
      with p↑ do
        begin clave := x; contador := 1;
          izquierdo := nil; derecho := nil; equi := 0
        end
    end
  end else
  if x < p↑ .clave then
    begin buscar (x, p↑ .izquierdo, h);
      if h then {la rama izquierda ha crecido}
        case p↑ .equi of
          1: begin p↑ .equi := 0; h := false
            end;
          0: p↑ .equi := -1;
          -1: begin {reequilibrar} p1 := p↑ .izquierdo;
            if p1↑ .equi = -1 then
              begin {rotacion II simple}
                p↑ .izquierdo := p1↑ .derecho; p1↑ .derecho := p;
                p↑ .equi := 0; p := p1
              end else
              begin {rotacion ID doble} p2 := p1↑ .derecho;
                p1↑ .derecho := p2↑ .izquierdo; p2↑ .izquierdo := p1;
                p↑ .izquierdo := p2↑ .derecho; p2↑ .derecho := p;
                if p2↑ .equi = -1 then p↑ .equi := +1 else p↑ .equi := 0;
                if p2↑ .equi = +1 then p1↑ .equi := -1 else p1↑ .equi := 0;
                p := p2
              end;
            p↑ .equi := 0; h := false
          end;
        end
      end
    end
  end else
  if x > p↑ .clave then
    begin buscar(x, p↑ .derecho, h);
      if h then {la rama derecha ha crecido}
        case p↑ .equi of
          1: begin p↑ .equi := 0; h := false
            end;

```

(4.63)

```

0: p↑ .equi := +1;
1: begin {reequilibrar} p1 := p↑ .derecho;
  if p1 .equi = +1 then
    begin {rotacion DD simple}
      p↑ .derecho := p1↑ .izquierdo; p1↑ .izquierdo := p;
      p↑ .equi := 0; p := p1
    end else
    begin {rotacion DI doble} p2 := p1↑ .izquierdo;
      p1↑ .izquierdo := p2↑ .derecho; p2↑ .derecho := p1;
      p↑ .derecho := p2↑ .izquierdo; p2↑ .izquierdo := p;
      if p2↑ .equi = +1 then p↑ .equi := -1 else p↑ .equi := 0;
      if p2↑ .equi = -1 then p1↑ .equi := +1 else p1↑ .equi := 0;
      p := p2
    end;
  p↑ .equi := 0; h := false
end
end
else
begin p↑ .contador := p↑ .contador + 1; h := false
end
end {buscar}

```

(4.63)

utilizar el cuarto caso de reequilibrado de (4.63), la rotación DI doble. El árbol final resultante se muestra en la Fig. 4.34 (f).

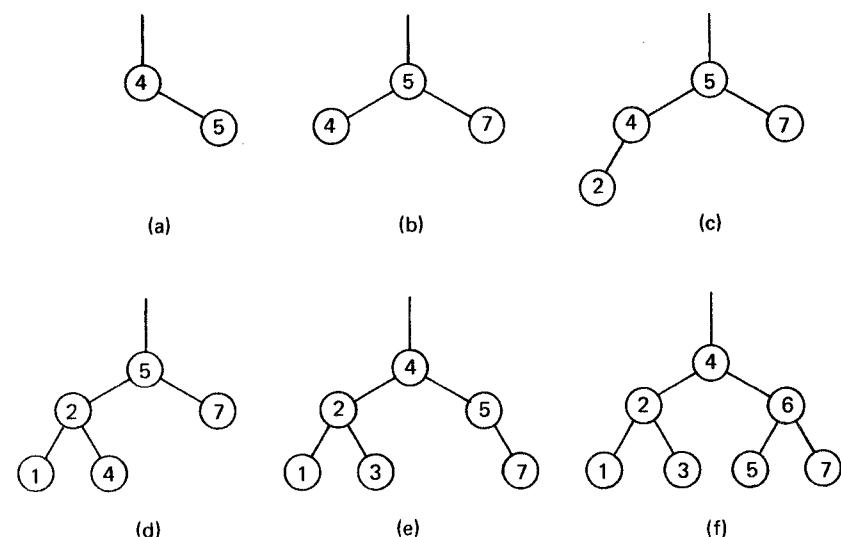


Fig. 4.34. Inserciones en un árbol equilibrado.

```

procedure buscar (x: integer; var p: ref; var h: boolean);
  var p1, p2: ref; {h = falso}
begin
  if p = nil then
    begin {la palabra no esta en el arbol; insertarla}
      new(p); h := true;
      with p↑ do
        begin clave := x; contador := 1;
          izquierdo := nil; derecho := nil; equi := 0
        end
    end else
      if x < p↑ .clave then
        begin buscar (x, p↑ .izquierdo, h);
          if h then {la rama izquierda ha crecido}
            case p↑ .equi of
              1: begin p↑ .equi := 0; h := false
                end;
              0: p↑ .equi := -1;
              -1: begin {reequilibrar} p1 := p↑ .izquierdo;
                  if p1↑ .equi = -1 then
                    begin {rotacion II simple}
                      p↑ .izquierdo := p1↑ .derecho; p1↑ .derecho := p;
                      p↑ .equi := 0; p := p1
                    end else
                      begin {rotacion ID doble} p2 := p1↑ .derecho;
                        p1↑ .derecho := p2↑ .izquierdo; p2↑ .izquierdo := p1;
                        p↑ .izquierdo := p2↑ .derecho; p2↑ .derecho := p;
                        if p2↑ .equi = -1 then p↑ .equi := +1 else p↑ .equi := 0;
                        if p2↑ .equi = +1 then p1↑ .equi := -1 else p1↑ .equi := 0;
                        p := p2
                      end;
                      p↑ .equi := 0; h := false
                    end;
                  end;
                end;
              end;
            end;
          end;
        end;
      else
        begin buscar (x, p↑ .derecho, h);
          if h then {la rama derecha ha crecido}
            case p↑ .equi of
              1: begin p↑ .equi := 0; h := false
                end;
              0: p↑ .equi := +1;
              -1: begin {reequilibrar} p1 := p↑ .derecho;
                  if p1↑ .equi = +1 then
                    begin {rotacion DD simple}
                      p↑ .derecho := p1↑ .izquierdo; p1↑ .izquierdo := p;
                      p↑ .equi := 0; p := p1
                    end else
                      begin {rotacion DI doble} p2 := p1↑ .izquierdo;
                        p1↑ .izquierdo := p2↑ .derecho; p2↑ .derecho := p1;
                        p↑ .derecho := p2↑ .izquierdo; p2↑ .izquierdo := p;
                        if p2↑ .equi = +1 then p↑ .equi := -1 else p↑ .equi := 0;
                        if p2↑ .equi = -1 then p1↑ .equi := +1 else p1↑ .equi := 0;
                        p := p2
                      end;
                      p↑ .equi := 0; h := false
                    end;
                  end;
                end;
              end;
            end;
          end;
        end;
      end;
    end;
  end;
end;

```

(4.63)

```

0: p↑ .equi := +1;
1: begin {reequilibrar} p1 := p↑ .derecho;
  if p1↑ .equi = +1 then
    begin {rotacion DD simple}
      p↑ .derecho := p1↑ .izquierdo; p1↑ .izquierdo := p;
      p↑ .equi := 0; p := p1
    end else
      begin {rotacion DI doble} p2 := p1↑ .izquierdo;
        p1↑ .izquierdo := p2↑ .derecho; p2↑ .derecho := p1;
        p↑ .derecho := p2↑ .izquierdo; p2↑ .izquierdo := p;
        if p2↑ .equi = +1 then p↑ .equi := -1 else p↑ .equi := 0;
        if p2↑ .equi = -1 then p1↑ .equi := +1 else p1↑ .equi := 0;
        p := p2
      end;
      p↑ .equi := 0; h := false
    end
  end;
else
  begin p↑ .contador := p↑ .contador + 1; h := false
  end
end {buscar}

```

(4.63)

utilizar el cuarto caso de reequilibrado de (4.63), la rotación *DI* doble. El árbol final resultante se muestra en la Fig. 4.34 (f).

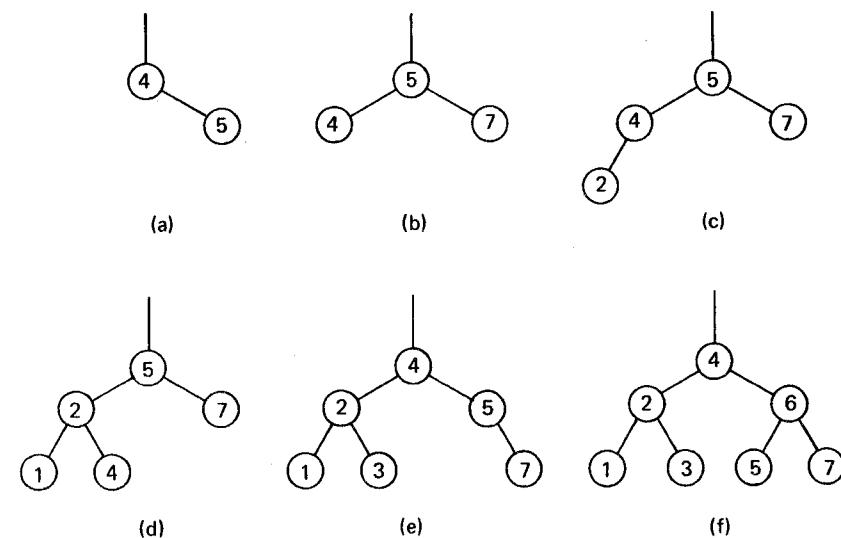


Fig. 4.34. Inserciones en un árbol equilibrado.

Dos preguntas especialmente interesantes sobre el rendimiento del algoritmo de inserción en un árbol equilibrado son las siguientes:

1. Si todas las $n!$ permutaciones de las n claves son igualmente probables, ¿cuál es la altura esperada del árbol equilibrado construido?
2. ¿Cuál es la probabilidad de que una inserción requiera reequilibrar el árbol?

El análisis matemático de este complicado algoritmo es todavía un problema no resuelto. Los ensayos empíricos apoyan la conjectura de que la altura esperada del árbol equilibrado construido por (4.63) es $h = \log(n) + c$, siendo c una constante pequeña ($c \approx 0,25$). Esto significa que, en la práctica, el árbol equilibrado AVL se comporta tan bien como el árbol perfectamente equilibrado, a pesar de que su estructura es mucho más fácil de mantener. La evidencia empírica también sugiere que, por término medio, se necesita reequilibrar el árbol cada dos inserciones, aproximadamente, y que las rotaciones simples y dobles son igualmente probables.* Evidentemente, el ejemplo de la Fig. 4.34 ha sido escogido cuidadosamente para ilustrar tantas rotaciones como fueran posibles con un número mínimo de inserciones, y no debe considerarse representativo.

La complejidad de las operaciones de equilibrado sugiere que estos árboles deben utilizarse sólo si las recuperaciones de información son considerablemente más frecuentes que las inserciones. Esto es especialmente cierto debido al hecho de que los nodos de tales árboles de búsqueda se codifican normalmente como registros densamente empaquetados con el fin de economizar memoria. Por lo tanto, la velocidad de acceso a, y actualización de, los factores de equilibrio

cada uno de los cuales sólo necesita dos bits—es a menudo un factor decisivo en la eficacia de la operación de reequilibrado. Las evaluaciones empíricas muestran que, si se necesita hacer empaquetado de registros, los árboles equilibrados pierden mucho de su interés. De hecho, ¡es difícil mejorar el algoritmo de inserción directa, simple, en árbol!

4.4.8. Borrado en árboles equilibrados

La experiencia adquirida en el caso de borrado en árboles sugiere que, en el caso de árboles equilibrados, ésta será también una operación más complicada que la inserción. Esto resulta ser cierto, a pesar de que la operación de reequilibrado es esencialmente la misma que en el caso de inserción. En particular, el reequilibrado es una rotación de nodos simple o doble.

El algoritmo básico de borrado en árbol equilibrado es el (4.52). Los casos fáciles son los nodos terminales y los nodos con un descendiente único. Si el nodo a borrar tiene dos subárboles, se le sustituirá por el más a la derecha del subárbol izquierdo. Tal como se hizo en el caso de inserción (4.63), se añade un parámetro booleano h que tiene el significado «la altura del subárbol ha disminuido». Solo se investiga si hace falta reequilibrar cuando h es cierto. Se asigna el valor verdadero a h cuando se encuentra y borra un nodo, o si al reequilibrar se reduce

la altura de un subárbol. En (4.64) se introducen dos operaciones (simétricas) de equilibrado; se formulan como procedimientos porque hay que llamarlas desde más de un punto del algoritmo de borrado. Obsérvese que *equilibrar1* se utiliza cuando la altura de la rama izquierda ha disminuido y *equilibrar2* cuando la de la rama derecha ha disminuido.

En la Fig. 4.35 se ilustra el funcionamiento del procedimiento. Dado el árbol equilibrado (a), borrando sucesivamente los nodos con claves 4, 8, 6, 5, 2, 1 y 7 se obtienen los árboles (b) . . . (h).

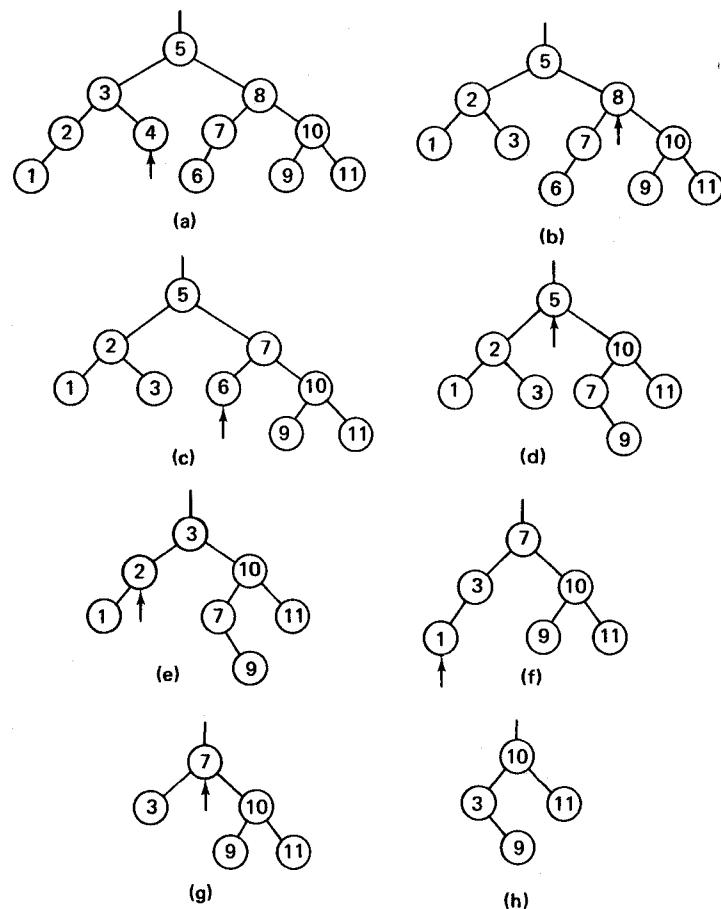


Fig. 4.35. Borrados en un árbol equilibrado.

El borrado de la clave 4 es simple en sí mismo, ya que es un nodo terminal. Sin embargo, se produce un desequilibrio en el nodo 3. Se reequilibra con una rotación II simple. Se necesita reequilibrar otra vez después de borrar el nodo 6.

Esta vez se reequilibra el subárbol derecho de la raíz (7) con una rotación *DD* simple. Al borrar el nodo 2, aunque en sí misma es una operación sencilla, ya que hay un sólo descendiente, se necesita hacer una rotación *DI* doble complicada. El cuarto caso, una rotación *ID* doble, se presenta, por último, después de borrar el nodo 7, que al principio fue sustituido por el elemento más a la derecha de un subárbol izquierdo, es decir, por el nodo con clave 3.

```

procedure borrar(x: integer; var p: ref; var h: boolean);
  var q: ref; {h = falso}
procedure equilibrar1(var p: ref; var h: boolean);
  var p1, p2: ref; e1, e2: -1 .. +1;
begin {h = cierto, la rama izquierda ha disminuido}
  case p↑ .equi of
    -1: p↑ .equi := 0;
    0: begin p↑ .equi := +1; h := false
      end;
    1: begin {reequilibrar} p1 := p↑ .derecho; e1 := p1 .equi;
      if e1 ≥ 0 then
        begin {rotacion DD simple}
          p↑ .derecho := p1↑ .izquierdo; p1↑ .izquierdo := p;
          if e1 = 0 then
            begin p↑ .equi := +1; p1↑ .equi := -1; h := false
            end else
              begin p↑ .equi := 0; p1↑ .equi := 0
              end;
          p := p1
        end else
          begin {rotacion DI doble}
            p2 := p1↑ .izquierdo; e2 := p2↑ .equi;
            p1↑ .izquierdo := p2↑ .derecho; p2↑ .derecho := p1;
            p↑ .derecho := p2↑ .izquierdo; p2↑ .izquierdo := p;
            if e2 = +1 then p↑ .equi := -1 else p↑ .equi := 0;
            if e2 = -1 then p1↑ .equi := +1 else p1↑ .equi := 0;
            p := p2; p2↑ .equi := 0
          end
        end
      end;
    end;
end {equilibrar1};

procedure equilibrar2(var p: ref; var h: boolean);
  var p1, p2: ref; e1, e2: -1 .. +1;
begin {h = cierto, la rama derecha ha disminuido}
  case p↑ .equi of
    1: p↑ .equi := 0;
    
```

(4.64)

```

0: begin p↑ .equi := -1; h := false
end;
-1: begin {reequilibrar} p1 := p↑ .izquierdo; e1 := p1↑ .equi;
  if e1 ≤ 0 then
    begin {rotacion II simple}
      p↑ .izquierdo := p1↑ .derecho; p1↑ .derecho := p;
      if e1 = 0 then
        begin p↑ .equi := -1; p1↑ .equi := +1; h := false
        end else
          begin p↑ .equi := 0; p1↑ .equi := 0
          end;
      p := p1
    end else
      begin {rotacion ID doble}
        p2 := p1↑ .derecho; e2 := p2↑ .equi;
        p1↑ .derecho := p2↑ .izquierdo; p2↑ .izquierdo := p1;
        p↑ .izquierdo := p2↑ .derecho; p2↑ .derecho := p;
        if e2 = -1 then p↑ .equi := +1 else p↑ .equi := 0;
        if e2 = +1 then p1↑ .equi := -1 else p1↑ .equi := 0;
        p := p2; p2↑ .equi := 0
      end
    end
  end {equilibrar2};

procedure bor(var r: ref; var h: boolean);
begin {h = falso}
  if r↑ .derecho ≠ nil then
    begin bor(r↑ .derecho, h); if h then equilibrar2(r, h)
    end else
      begin r↑ .clave := r↑ .clave; q↑ .contador := r↑ .contador;
        r := r↑ .izquierdo; h := true
      end
  end;

begin {borrar}
  if p = nil then
    begin writeln ('LA CLAVE NO ESTA EN EL ARBOL'); h := false
    end else
      if x < p↑ .clave then
        begin borrar(x, p↑ .izquierdo, h); if h then equilibrar1(p, h)
        end else
          if x > p↑ .clave then
            begin borrar(x, p↑ .derecho, h); if h then equilibrar2(p, h)
            end else

```

(4.64)

```

begin {borrar} p↑} q := p;
  if q↑ .derecho = nil then
    begin p := q↑ .izquierdo; h := true
    end else
  if q↑ .izquierdo = nil then
    begin p := q↑ .derecho; h := true
    end else
  begin bor(q↑ .izquierdo, h);
    if h then equilibrar1(p, h)
  end;
  {dispose(q)}
end
end {borrar}

```

(4.64)

Evidentemente, se puede borrar un elemento en un árbol equilibrado con en el caso más desfavorable— $O(\log n)$ operaciones. Sin embargo, no debe pasarse por alto una diferencia esencial que existe entre los procedimientos de inserción y borrado. Mientras, al realizar una inserción de una sola clave, se puede producir como máximo una rotación (de dos o tres nodos), el borrado puede requerir una rotación en *todos* los nodos del camino de búsqueda. Considerese, por ejemplo, el borrado del nodo más a la derecha en un árbol Fibonacci. En este caso, el borrado de un único nodo supone una reducción de la altura del árbol; además, el borrado del nodo más a la derecha requiere el máximo número de rotaciones. Pero esto significa elegir el peor nodo en el peor tipo de árbol equilibrado, una combinación que será bastante improbable en la práctica. Entonces, ¿con qué frecuencia se presentan las rotaciones en general?

Sorprendentemente, los análisis empíricos dan como resultado que, mientras se presenta una rotación por cada dos inserciones, aproximadamente, sólo se necesita una por cada cinco borrados. El borrado en árboles equilibrados es, pues, tan sencillo —o tan complicado— como la inserción.

4.4.9. Arboles de búsqueda óptimos

Hasta ahora, todas las consideraciones realizadas sobre la forma de organizar árboles de búsqueda, se han hecho sobre la hipótesis básica de que todos los nodos tienen la misma frecuencia de acceso. Esta es probablemente la mejor hipótesis que se puede hacer cuando no se conoce la distribución de accesos. Sin embargo, existen casos (son la excepción más que la regla) en los que se dispone de información sobre las probabilidades de acceso a los nodos individuales. Es normalmente una característica de estos casos que las claves se mantienen siempre las mismas, es decir, no se producen inserciones ni borrados en el árbol de búsqueda, y éste mantiene una estructura constante. Un ejemplo típico es el analizador léxico de un compilador que determina si una palabra (identificador) es una palabra clave (palabra reservada) o no. En este caso, se puede obtener

información bastante aproximada sobre la frecuencia relativa con que se presentan las claves individuales y, por lo tanto, sobre su frecuencia de acceso, realizando medidas estadísticas en la compilación de cientos de programas.

Supóngase que, en un árbol de búsqueda, se accede el nodo i con una probabilidad p_i .

$$\Pr\{x = k_i\} = p_i, \quad \sum_{i=1}^n p_i = 1 \quad (4.65)$$

Se desea ahora organizar el árbol de búsqueda de tal manera que el número total de pasos de búsqueda —contados con un número de ensayos suficientemente alto— sea mínimo. A tal fin, se modifica la definición de la longitud de camino (4.34) asignando un cierto peso a cada nodo. Los nodos con una frecuencia de acceso alta se convierten en nodos «pesados»; aquéllos que son visitados poco frecuentemente en nodos «ligeros». La *longitud de camino* (interno) ponderada es entonces la suma de todos los caminos de la raíz a cada nodo multiplicados por la probabilidad de acceso de cada nodo.

$$C_I = \sum_{i=1}^n p_i h_i \quad (4.66)$$

h_i es el nivel del nodo i (o su distancia a la raíz + 1). El objetivo ahora es *minimizar la longitud de camino ponderada*, dada una cierta distribución de las probabilidades.

Como ejemplo, considérese el conjunto de claves 1, 2, 3, con probabilidades de acceso $p_1 = 1/7$, $p_2 = 2/7$, y $p_3 = 4/7$. Estas claves pueden organizarse como árboles de búsqueda de 5 maneras diferentes (ver Fig. 4.36).

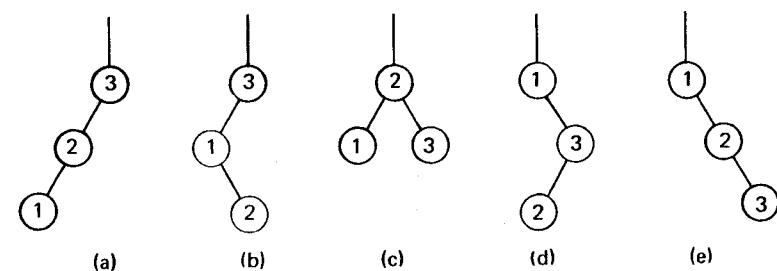


Fig. 4.36. Arboles de búsqueda con tres nodos.

Las longitudes de camino ponderadas, calculadas según (4.66) son:

$$C_I^{(a)} = \frac{1}{7}(1 \cdot 3 + 2 \cdot 2 + 4 \cdot 1) = \frac{11}{7}$$

$$C_I^{(b)} = \frac{1}{7}(1 \cdot 2 + 2 \cdot 3 + 4 \cdot 1) = \frac{12}{7}$$

$$C_I^{(c)} = \frac{1}{7}(1 \cdot 2 + 2 \cdot 1 + 4 \cdot 2) = \frac{12}{7}$$

$$C_I^{(d)} = \frac{1}{7}(1 \cdot 1 + 2 \cdot 3 + 4 \cdot 2) = \frac{15}{7}$$

$$C_I^{(e)} = \frac{1}{7}(1 \cdot 1 + 2 \cdot 2 + 4 \cdot 3) = \frac{17}{7}$$

Por lo tanto, en este ejemplo el árbol perfectamente equilibrado no es la organización óptima, sino el árbol degenerado (a).

El ejemplo del analizador léxico sugiere inmediatamente que el problema debe contemplarse con una óptica más amplia, ya que las palabras que aparecen en el texto no siempre son palabras clave; de hecho, el que sean palabras clave será más bien una excepción. Descubrir que una palabra dada k no es una clave del árbol de búsqueda puede considerarse como el acceso a un «nodo especial» hipotético insertado entre las claves inmediatamente más baja y más alta que la clave dada (ver Fig. 4.19), con una cierta longitud de camino externo. Si se conoce también la probabilidad q_i de que un argumento a buscar se encuentre entre las claves k_i y k_{i+1} , esta información puede cambiar considerablemente la estructura del árbol de búsqueda óptimo. Por lo tanto, se generaliza el problema para tener en cuenta también las búsquedas infructuosas.

La longitud de camino media ponderada total es ahora

$$C = \sum_{i=1}^n p_i h_i + \sum_{j=0}^m q_j h'_j \quad (4.67)$$

siendo

$$\sum_{i=1}^n p_i + \sum_{j=0}^m q_j = 1$$

h_i es el nivel del nodo (interno) i y h'_j es el nivel del nodo externo j . La longitud de camino media ponderada puede llamarse el «coste» del árbol de búsqueda, ya que representa una medida del esfuerzo que, por término medio, hará falta realizar en la búsqueda. El árbol de búsqueda cuya estructura da el mínimo coste, entre todos los árboles con un conjunto dado de claves k_i y probabilidades p_i y q_i , recibe el nombre de *árbol óptimo*.

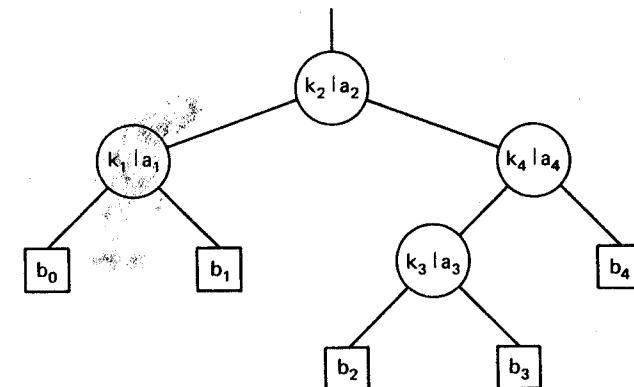


Fig. 4.37. Árbol de búsqueda con sus frecuencias de acceso.

Para encontrar el árbol óptimo, no hace falta que las p y las q sumen la unidad. De hecho, estas probabilidades se determinan frecuentemente por medio de experimentos en los que se cuenta el número de accesos a los nodos. En vez de utilizar las probabilidades p_i y q_i , se utilizarán en lo que sigue números de acceso totales, designados por

a_i = número de veces que el argumento de búsqueda x es igual a k_i

b_j = número de veces que el argumento de búsqueda x se encuentra entre k_j y k_{j+1}

Se conviene en que b_0 es el número de veces que x es menor que k_1 y b_n el número de veces que x es mayor que k_n (ver Fig. 4.37). En lo que sigue se utilizará C para designar la longitud de camino ponderada acumulada, en vez de la longitud de camino media:

$$C = \sum_{i=1}^n a_i h_i + \sum_{j=0}^n b_j h'_j \quad (4.68)$$

De esta manera, además de no tener que calcular las probabilidades de acceso a partir del número de ellos, se consigue la ventaja adicional de poder buscar el árbol óptimo utilizando nada más que números enteros.

Considerando que el número de configuraciones posibles de n nodos crece exponencialmente, parece poco menos que imposible poder encontrar el árbol óptimo cuando n es grande. Los árboles óptimos, sin embargo, tienen una importante propiedad que ayuda a encontrarlos: todos sus subárboles son también óptimos. Por ejemplo, si el árbol de la Fig. 4.37 es óptimo para ciertas a y b dadas, entonces el subárbol con claves k_3 y k_4 , de la misma figura, también es óptimo. Esta propiedad sugiere un algoritmo que, de una forma sistemática, encuentra

árboles cada vez mayores, partiendo de nodos individuales como los subárboles más pequeños posibles. De esta manera, el árbol crece «desde las ramas hacia la raíz» lo que significa, puesto que se están dibujando los árboles dados la vuelta, la dirección «ascendente» [4.6].

La ecuación (4.69) es la clave de este algoritmo. Sea C la longitud de camino ponderada de un árbol, y C_I y C_D las de los subárboles izquierdo y derecho de su raíz. Está claro que C es igual a la suma de C_I , C_D y el número de veces que una búsqueda viaja sobre el arco único que va a la raíz, lo cual es, simplemente, el número total de búsquedas P .

$$C = C_I + P + C_D \quad (4.69)$$

$$P = \sum_{i=1}^n a_i + \sum_{j=0}^n b_j \quad (4.70)$$

P se denomina *peso* del árbol. Su longitud de camino *media* es entonces C/P .

Estas consideraciones muestran la necesidad de disponer de una notación para los pesos y las longitudes de camino de cualquier subárbol formado por un conjunto adyacente de claves.

Sea p_{ij} el peso del subárbol óptimo A_{ij} formado por los nodos con claves $k_{i+1}, k_{i+2}, \dots, k_j$, y c_{ij} su longitud de camino. Estas cantidades se definen mediante las relaciones recurrentes (4.71) y (4.72).

$$p_{ii} = b_i \quad (0 \leq i \leq n) \quad (4.71)$$

$$p_{ij} = p_{i,j-1} + a_j + b_j \quad (0 \leq i < j \leq n)$$

$$c_{ii} = p_{ii} \quad (0 \leq i \leq n) \quad (4.72)$$

$$c_{ij} = p_{ij} + \min_{i < k \leq j} (c_{i,k-1} + c_{kj}) \quad (0 \leq i < j \leq n)$$

La última ecuación se deduce inmediatamente a partir de (4.69) y la definición de árbol óptimo.

Como existen, aproximadamente $(1/2)n^2$ valores p_{ij} , y (4.72) requiere una elección entre $0 < j - i \leq n$ casos, la operación de minimización necesitará del orden de $(1/6)n^3$ operaciones. Knuth ha indicado cómo puede eliminarse un factor n lo que, por sí solo, salva este algoritmo desde un punto de vista práctico.

Sea r_{ij} el valor de k que produce el valor mínimo en (4.72). Puede limitarse la búsqueda de r_{ij} a un intervalo mucho más pequeño, es decir, reducir el número de pasos de evaluación $j - i$. La clave está en observar que, si ya se ha encontrado r_{ij} , raíz del subárbol óptimo A_{ij} , tanto al ampliar el árbol añadiendo un nodo a su derecha, como al sacar su nodo más a la izquierda, la raíz no se desplaza a la izquierda. Esto se expresa mediante la relación

$$r_{i,i-1} \leq r_{ij} \leq r_{i+1,j} \quad (4.73)$$

que limita la búsqueda de posibles soluciones de r_{ij} al campo $r_{i,j-1} \dots r_{i+1,j}$, con el resultado de que se presentan un número total de pasos elementales del orden $O(n^2)$. Ahora se está en disposición de poder construir el algoritmo de optimización en detalle. Recuérdense las siguientes definiciones relativas a árboles óptimos A_{ij} formados por nodos con claves $k_{i+1} \dots k_j$.

1. a_i : el número de veces que se busca k_i .
2. b_j : el número de veces que el argumento buscado x se encuentra entre k_j y k_{j+1} .
3. p_{ij} : el peso de A_{ij} .
4. c_{ij} : la longitud de camino ponderada de A_{ij} .
5. r_{ij} : el índice de la raíz de A_{ij} .

Dado

type indice = 0 .. n

se declaran los arrays siguientes:

```
a: array[1 .. n] of integer;
b: array[indice] of integer;
c, p: array[indice, indice] of integer;
r: array[indice, indice] of indice
```

(4.74)

Se supone que los pesos p_{ij} han sido calculados a partir de a y b de forma directa [ver (4.71)]. Considérese ahora p como el argumento de un procedimiento que se va a desarrollar y r como su resultado, ya que r describe la estructura completamente. c puede considerarse un resultado intermedio. Empezando con los subárboles más pequeños posibles, aquéllos que no tienen ningún nodo, se construyen árboles cada vez más grandes.

Sea h la anchura $j - i$ del subárbol A_{ij} . La determinación de los valores c_{ii} de todos los árboles con anchura $h = 0$ es trivial; aplicando (4.72) se tiene

for $i := 0$ to n do $c[i, i] := p[i, i]$ (4.75)

En el caso de $h = 1$, se trata de árboles formados por un solo nodo, que también es la raíz (Fig. 4.38).

```
for  $i := 0$  to  $n - 1$  do
begin  $j := i + 1$ ;  $c[i, j] := c[i, i] + c[j, j]$ ;  $r[i, j] := j$ 
end
```

(4.76)

Obsérvese que i significa el límite izquierdo del índice, y j el límite derecho del mismo, en el árbol considerado A_{ij} . Para los casos $h > 1$ se utiliza una instrucción

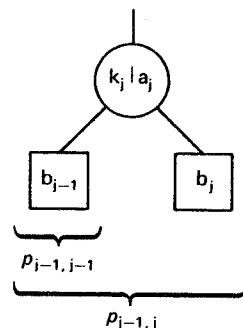


Fig. 4.38. Arbol óptimo de un único nodo.

repetitiva con h variando desde 2 a n ; cuando $h = n$ se tiene el árbol completo $A_{0,n}$. En cada caso, se determinan la longitud de camino mínima c_{ij} y el correspondiente índice de la raíz r_{ij} haciendo uso de una simple instrucción repetitiva, con un índice k que cubre el intervalo dado por (4.73).

```

for h := 2 to n do
  for i := 0 to n - h do
    begin j := i + h;
      «encontrar m y min = mínimo (c[i, m - 1] + c[m, j]) para
       m tal que r[i, j - 1] ≤ m ≤ r[i + 1, j]»;
      c[i, j] := min + p[i, j]; r[i, j] := m
    end
  
```

(4.77)

En el Programa 4.6 se pueden encontrar los detalles de la instrucción que se presenta entre comillas. La longitud de camino media de $A_{0,n}$ será el cociente $c_{0,n}/p_{0,n}$ y su raíz el nodo con índice $r_{0,n}$.

Resulta evidente, a partir del algoritmo (4.77), que el esfuerzo que se requiere para determinar la estructura óptima es $O(n^2)$; la cantidad de memoria necesaria es también $O(n^2)$. Esto es inaceptable si n es muy grande. Por lo tanto, es de gran interés conseguir algoritmos que tengan mayor eficacia. Uno de ellos ha sido desarrollado por Hu y Tucker [4-5] y requiere solamente $O(n)$ memoria y $O(n \cdot \log n)$ computaciones. Este algoritmo, sin embargo, considera sólo el caso en que el número de veces que se buscan las claves es cero ($a_i = 0$); es decir, el caso en el que sólo se presentan búsquedas infructuosas. Otro algoritmo, que también utiliza $O(n)$ elementos de memoria y $O(n \cdot \log n)$ computaciones, ha sido descrito por Walker y Gotlieb [4-11]. Este algoritmo, en vez de buscar el árbol óptimo, simplemente promete encontrar un árbol casi óptimo. Puede, por lo tanto, ser basado en principios *heurísticos*. La idea básica es la siguiente.

Considérense los nodos (incluyendo los nodos especiales) distribuidos según una línea recta, con pesos iguales a sus números (o frecuencias) de acceso. A continuación se busca el nodo que está más cerca del «centro de gravedad». Este nodo se denomina «*centroide*» y su índice es:

$$\frac{1}{p} \left(\sum_{i=1}^n i \cdot a_i + \sum_{j=0}^n j \cdot b_j \right) \quad (4.78)$$

redondeado al entero más cercano. Si todos los nodos tienen el mismo peso, entonces el centroide coincide con la raíz del árbol óptimo, evidentemente, y —continúa el razonamiento— dicha raíz estará, en la mayor parte de los casos, muy cerca del centroide. Así pues, se aplica una búsqueda limitada para encontrar el óptimo local, y a continuación se aplica el mismo proceso a los dos subárboles resultantes. La probabilidad de que la raíz esté muy cercana al centroide crece con el tamaño del árbol n . Tan pronto como los subárboles han alcanzado un tamaño «manejable», se puede encontrar su óptimo aplicando el algoritmo anterior que es exacto.

4.4.10. Presentación de estructuras árbol

Se considera ahora el problema de cómo programar la generación de una impresión que presente la estructura de un árbol de una forma gráfica, suficientemente clara, utilizando solamente una impresora común. Es decir, se quiere dibujar una imagen de un árbol, imprimiendo las claves como nodos y conectándoles por medio de caracteres «barra» vertical y horizontal apropiados.

Una impresora tiene sus datos definidos como un fichero de texto, es decir, como una secuencia de caracteres, y se puede avanzar en ella únicamente en dirección de izquierda a derecha y de arriba abajo. Por lo tanto, parece razonable construir primero una representación del *árbol* que refleje apropiadamente su estructura topológica. El segundo paso es *aplicar* de una forma ordenada esta imagen sobre la página impresa y calcular las coordenadas exactas de los nodos y arcos.

Para la primera parte se puede utilizar la experiencia adquirida con los algoritmos de generación de árboles y, sin dudarlo, elegir una solución recursiva, ya que el problema está definido recursivamente. Se formula un procedimiento función llamado *arbol* similar al utilizado en el Programa 4.3. Los parámetros i, j son los índices límite de los nodos que pertenecen al árbol. Su raíz se define entonces como el nodo con índice r_{ij} . Antes de continuar, sin embargo, hace falta definir el tipo de las variables que van a representar los nodos. Estos deben contener los dos punteros a los subárboles y la clave del nodo. Se utilizan dos campos adicionales, llamados *pos* y *enlace*, con fines que se justificarán en la segunda fase. En (4.79) se muestran las definiciones adoptadas y el procedimiento función resultante aparece en el Programa 4.6.

```

type ref = ↑nodo;
nodo == record clave: alfa;
           pos: poslinea;
           izquierdo,'derecho, enlace: ref
         end
  
```

(4.79)

Obsérvese que este procedimiento cuenta el número de nodos generados, con la variable auxiliar k . El nodo k -ésimo recibe la clave k -ésima y, como las claves están ordenadas alfabéticamente, k multiplicado por una constante da la coordenada horizontal de cada clave, valor que es almacenado inmediatamente en unión del resto de la información. Obsérvese también que no se ha utilizado el convenio de usar enteros como claves, y se les supone del tipo *alfa*, significando éste un array de caracteres, de una longitud (máxima) dada, sobre el que se encuentra definido un orden alfabético. Para poner en evidencia lo conseguido hasta ahora, obsérvese la Fig. 4.39. Dados el conjunto de n claves y la matriz calculada r_{ij} , las instrucciones

$$k := 0; \quad raiz := arbol(0, n)$$

generarán la estructura preliminar del árbol enlazado con las porciones horizontales de cada nodo anotadas en el campo *pos*, y sus posiciones verticales determinadas implícitamente por su nivel en el árbol.

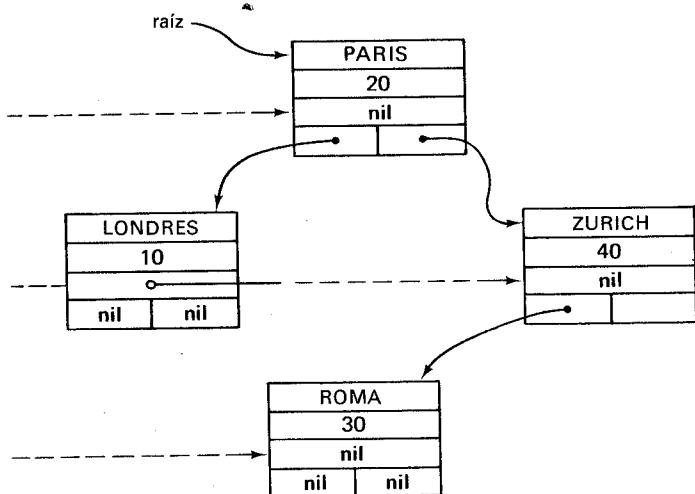


Fig. 4.39. Arbol resultado del Programa 4.6.

Ahora se está en condiciones de analizar la segunda parte: dibujar el árbol en el papel. En este caso, debe avanzarse de una forma estrictamente secuencial, partiendo del nivel de la raíz hacia abajo, procesando en cada paso una fila de nodos. Pero, ¿cómo se llega a los nodos que se encuentran en una fila? Para ello, para enlazar entre sí los nodos que están en la misma fila, se ha introducido previamente el campo llamado *enlace* en el registro de un nodo. En la Fig. 4.39 se muestran con líneas de trazo las cadenas que hay que establecer. En cada paso del proceso se supone la presencia de una cadena, que une los nodos a imprimir,

denominada cadena *encurso*, y al procesar cada nodo se identifican sus descendientes (si los hay), y se les enlaza según una segunda cadena —llamada cadena *siguiente*—. Cuando se avanza hacia abajo un nivel, la cadena siguiente se convierte en la cadena en curso, y la cadena siguiente se vuelve a inicializar como vacía.

Los detalles del algoritmo pueden ser tomados del Programa 4.6. Las notas siguientes pueden ayudar a aclarar algunos puntos:

1. Las cadenas de nodos de una fila se generan de izquierda a derecha, con el resultado de que el nodo más a la izquierda queda el último en la cadena. Como los nodos deben ser visitados de izquierda a derecha, se ha de invertir la lista. Esta inversión se realiza cuando la lista *siguiente* se convierte en la lista *encurso*.
2. Una línea impresa que contenga las claves —llamada línea maestra— también contiene los arcos horizontales (ver Fig. 4.40). Las variables u_1 , u_2 , u_3 , u_4 denotan las posiciones del principio y final de los arcos horizontales izquierdo y derecho de un nodo.
3. La construcción de cada línea maestra está precedida de tres líneas que marcan las partes verticales de los arcos.

A continuación se describe la estructura del Programa 4.6. Sus dos partes principales son los procedimientos para encontrar el árbol de búsqueda óptimo, dada una cierta distribución de pesos p , y para dibujar el árbol, dados los índices r . El programa completo está adaptado para procesar textos de programas, en particular, programas PASCAL. En la primera parte se lee un programa y se reconocen sus identificadores y palabras clave, calculando los valores a_i y b_j para claves k_i e identificadores entre k_j y k_{j+1} . Despues de imprimir las estadísticas de aparición de palabras, el programa continúa y calcula la longitud de camino del árbol perfectamente equilibrado, determinando al mismo tiempo las raíces de sus subárboles. A continuación se imprime la longitud de camino media y se dibuja el árbol.

En la tercera parte se ejecuta el procedimiento *arbolopt* para calcular el árbol de búsqueda óptimo; a continuación, se dibuja también este último. Finalmente, se utilizan los mismos procedimientos para calcular y dibujar el árbol óptimo considerando únicamente las frecuencias de aparición de las claves.

La Tabla 4.5 y las Figs. 4.40 y 4.42 muestran los resultados generados por el Programa 4.6 cuando se aplica a su propio texto*. Las diferencias en los tres valores demuestran que el árbol equilibrado no puede ser considerado ni siquiera como casi óptimo, y que las frecuencias de aparición de las palabras que no son clave tienen una influencia crucial en la elección de la estructura óptima.

* N. del T. Más concretamente, cuando se aplica al texto del programa en su versión inglesa, sin traducir los identificadores. Las frecuencias de las palabras clave (valores de las a) son las mismas para ambas versiones, pero los valores de las b varían. Se ha optado por presentar el programa con los símbolos modificados respecto de la versión inglesa, como el resto de los programas del libro, por considerarse más interesante facilitar la legibilidad del programa que la precisión, anecdótica, de los resultados.

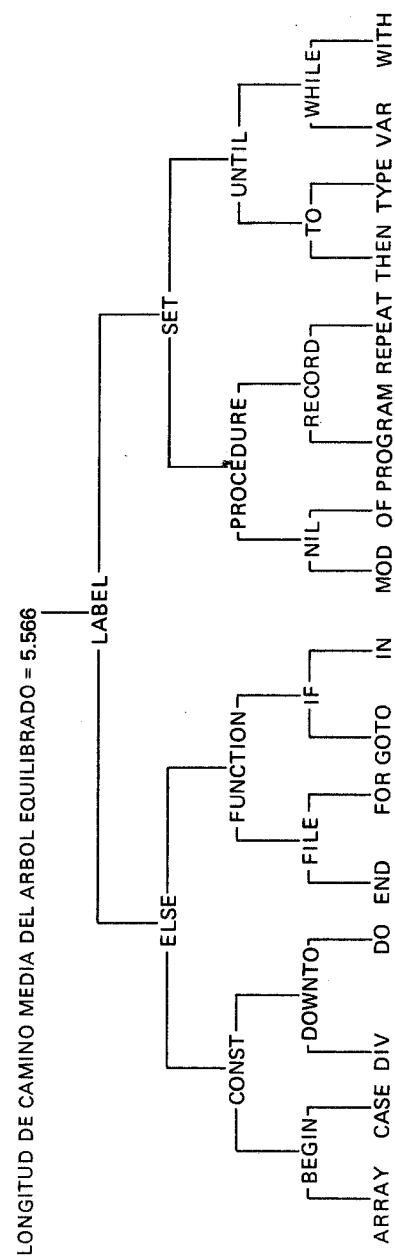


Fig. 4.40. Arbol perfectamente equilibrado.

4	7 ARRAY
14	27 BEGIN
19	0 CASE
15	2 CONST
8	5 DIV
0	0 DOWNT0
0	20 DO
0	8 ELSE
0	28 END
1	0 FILE
0	12 FOR
0	2 FUNCTION
0	0 GOTO
9	13 IF
23	2 IN
208	0 LABEL
22	0 MOD
17	10 NIL
24	7 OF
17	2 PROCEDURE
0	1 PROGRAM
53	1 RECORD
6	8 REPEAT
16	0 SET
10	13 THEN
0	12 TO
6	2 TYPE
1	8 UNTIL
39	5 VAR
0	8 WHILE
0	0 WITH
37	
549	203

Tabla 4.5. Claves y número de veces que aparecen.

LONGITUD DE CAMINO MEDIA DEL ARBOL OPTIMO = 4.160

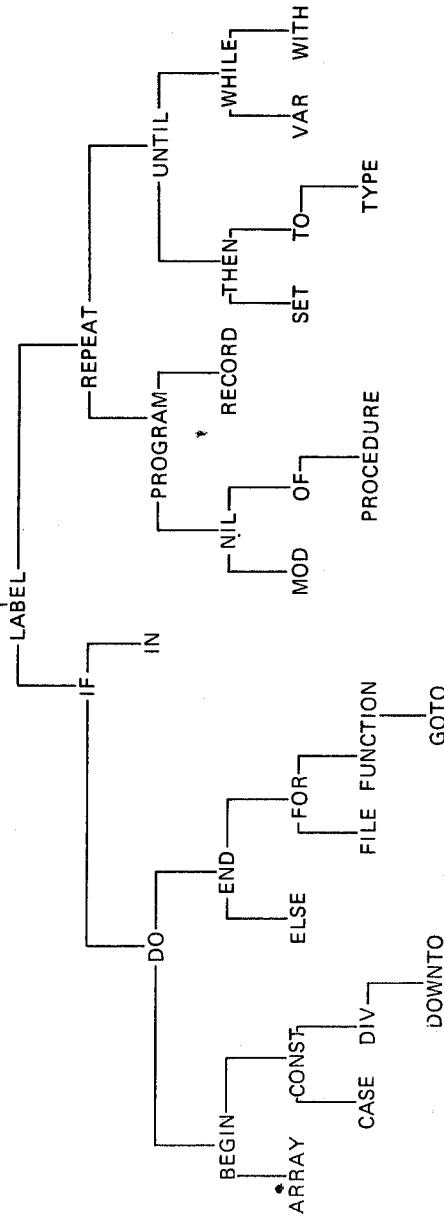


Fig. 4.41. Árbol de búsqueda óptimo.

ARBOL OPTIMO CONSIDERANDO SOLO LAS CLAVES

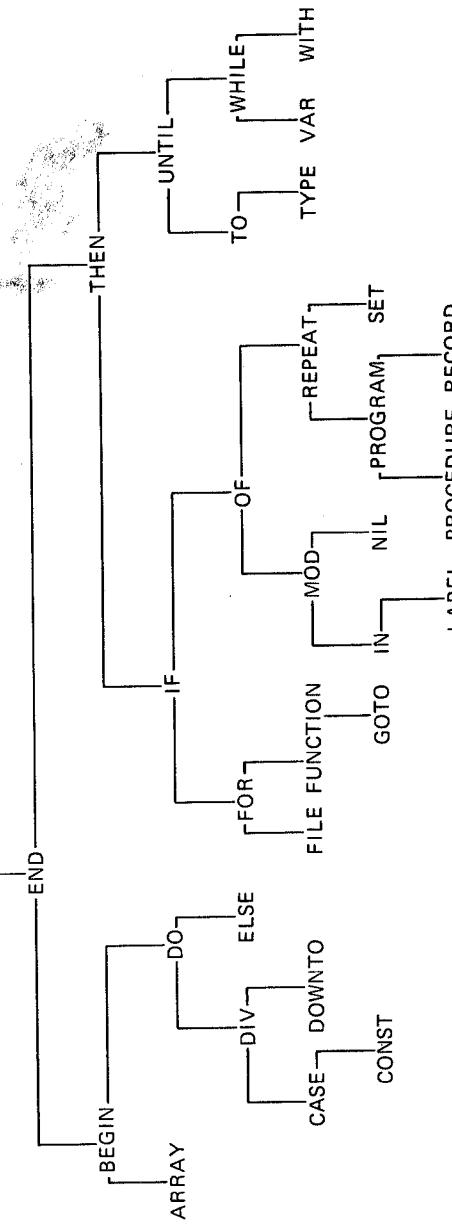


Fig. 4.42. Árbol óptimo considerando sólo las claves.

Programa 4.6. Encontrar árbol de búsqueda óptimo.

```

program arboloptimo(input, output);
const n = 31; {numero de claves}
  loncl = 10; {longitud maxima de la clave}
type indice = 0 .. n;
alfa = packed array [1 .. loncl] of char;
var ch: char;
  k1, k2: integer;
  id: alfa; {identificador o clave}
  buf: array [1 .. loncl] of char; {buffer de caracteres}
  clave: array [1 .. n] of alfa;
  i, j, k: integer;
  a: array [1 .. n] of integer;
  b: array [indice] of integer;
  c, p: array [indice, indice] of integer;
  r: array [indice, indice] of indice;
  suma, sumb: integer;
function arbolequi(i, j: indice): integer;
  var k: integer;
begin k := (i + j + 1) div 2; r[i, j] := k;
  if i ≥ j then arbolequi := b[k] else
    arbolequi := arbolequi(i, k - 1) + arbolequi(k, j) + p[i, j]
end {arbolequi};
procedure arbolopt;
  var x, min: integer;
  i, j, k, h, m: indice;
begin {argumento: p, resultado: c, r}
  for i := 0 to n do c[i, i] := p[i, i]; {ancho del arbol, h = 0}
  for i := 0 to n - 1 do {ancho del arbol, h = 1}
  begin j := i + 1;
    c[i, j] := c[i, i] + c[j, j]; r[i, j] := j
  end;
  for h := 2 to n do {h = ancho del arbol considerado}
  for i := 0 to n - h do {i = indice izquierdo del arbol considerado}
  begin j := i + h; {j = indice derecho del arbol considerado}
    m := r[i, j - 1]; min := c[i, m - 1] + c[m, j];
    for k := m + 1 to r[i + 1, j] do
      begin x := c[i, k - 1] + c[k, j];
        if x < min then
          begin m := k; min := x
          end
      end;
    c[i, j] := min + p[i, j]; r[i, j] := m
  end;
end {arbolopt};

```

```

procedure imprimirarbol;
  const al = 120; {ancho de linea de la impresora}
  type ref: ↑nodo;
    poslinea: 0 .. al;
    nodo = record clave: alfa;
      pos: poslinea;
      izquierdo, derecho, enlace: ref
    end;
  var raiz, encurso, siguiente: ref;
    q, q1, q2: ref;
    i, k: integer;
    u, u1, u2, u3, u4: poslinea;
  function arbol(i, j: indice): ref;
    var p: ref;
  begin if i = j then p := nil else
    begin new(p);
      p↑ .izquierdo := arbol(i, r[i, j] - 1);
      p↑ .pos := trunc((al-loncl) * k/(n - 1)) + (loncl div 2); k := k + 1;
      p↑ .clave := clave[r[i, j]];
      p↑ .derecho := arbol(r[i, j], j)
    end;
    arbol := p
  end;
  begin k := 0; raiz := arbol(0, n);
  encurso := raiz; raiz↑ .enlace := nil;
  siguiente := nil;
  while encurso ≠ nil do
  begin {avanzar hacia abajo; primero escribir las lineas verticales}
    for i := 1 to 3 do
    begin u := 0; q := encurso;
      repeat u1 := q↑ .pos;
        repeat write(' ');
        until u = u1;
        write('|');
        u := u + 1; q := q↑ .enlace
      until q = nil;
      writeln
    end;
    {ahora se imprime la linea maestra; a partir de los nodos de la lista
    en curso, se agrupan sus descendientes para formar la lista siguiente}
  end;

```

Programa 4.6. (Continuación)

```

 $q := encuso; u := 0;$ 
repeat unpack( $q \uparrow . clave, buf, 1$ );
{centrar la clave en pos}  $i := loncla;$ 
while  $buf[i] = ' '$  do  $i := i - 1$ ;
 $u2 := q \uparrow . pos - ((i - 1) \text{ div } 2); u3 := u2 + i$ ;
 $q1 := q \uparrow . izquierdo; q2 := q \uparrow . derecho;$ 
if  $q1 = \text{nil}$  then  $u1 := u2$  else
begin  $u1 := q1 \uparrow . pos; q1 \uparrow . enlace := siguiente; siguiente := q1$ 
end;
if  $q2 = \text{nil}$  then  $u4 := u3$  else
begin  $u4 := q2 \uparrow . pos + 1; q2 \uparrow . enlace := siguiente; siguiente := q2$ 
end;
 $i := 0$ ;
while  $u < u1$  do begin write(' ');  $u := u + 1$  end;
while  $u < u2$  do begin write(' -');  $u := u + 1$  end;
while  $u < u3$  do begin  $i := i + 1$ ; write(buf[i]);  $u := u + 1$  end;
while  $u < u4$  do begin write(' -');  $u := u + 1$  end;
 $q := q \uparrow . enlace$ 
until  $q = \text{nil}$ ;
writeln;
{ahora se invierte la lista siguiente y se le hace ser la lista en curso}
encuso := nil;
while siguiente ≠ nil do
begin  $q := siguiente; siguiente := q . enlace;$ 
 $q \uparrow . enlace := encuso; encuso := q$ 
end
end
end {imprimirarbol};
begin {inicializar tabla de claves y contadores}

clave[ 1] := 'ARRAY'; clave[ 2] := 'BEGIN';
clave[ 3] := 'CASE'; clave[ 4] := 'CONST';
clave[ 5] := 'DIV';
clave[ 7] := 'DO';
clave[ 9] := 'END';
clave[11] := 'FOR';
clave[13] := 'GOTO';
clave[15] := 'IN';
clave[17] := 'MOD';
clave[19] := 'OF';
clave[21] := 'PROGRAM';
clave[ 2] := 'BEGIN';
clave[ 4] := 'CONST';
clave[ 6] := 'DOWNTO';
clave[ 8] := 'ELSE';
clave[10] := 'FILE';
clave[12] := 'FUNCTION';
clave[14] := 'IF';
clave[16] := 'LABEL';
clave[18] := 'NIL';
clave[20] := 'PROCEDURE';
clave[ 22] := 'RECORD';

```

Programa 4.6. (Continuación)

```

clave[23] := 'REPEAT'; clave[24] := 'SET';
clave[25] := 'THEN'; clave[26] := 'TO';
clave[27] := 'TYPE';
clave[29] := 'VAR';
clave[31] := 'WITH';
for  $i := 1$  to  $n$  do
begin  $a[i] := 0; b[i] := 0$ 
end;
 $b[0] := 0; k2 := loncla$ ;
{examinar el texto de entrada y determinar a y b}
while  $\neg eof(input)$  do
begin read(ch);
if ch in ['A' .. 'Z'] then
begin {identificador o clave}  $k1 := 0$ ;
repeat if  $k1 < loncl$  then
begin  $k1 := k1 + 1; buf[k1] := ch$ 
end;
read(ch)
until  $\neg ch$  in ['A' .. 'Z', '0' .. '9'];
if  $k1 \geq k2$  then  $k2 := k1$  else
repeat  $buf[k2] := ' '$ ;  $k2 := k2 - 1$ 
until  $k2 = k1$ ;
pack(buf, 1, id);
 $i := 1; j := n$ ;
repeat  $k := (i + j) \text{ div } 2$ ;
if  $clave[k] \leq id$  then  $i := k + 1$ ;
if  $clave[k] \geq id$  then  $j := k - 1$ ;
until  $i > j$ ;
if  $clave[k] = id$  then  $a[k] := a[k] + 1$  else
begin  $k := (i + j) \text{ div } 2; b[k] := b[k] + 1$ 
end
end else
if  $ch = " "$  then
repeat read(ch) until  $ch = " "$  else
if  $ch = '{'$  then
repeat read(ch) until  $ch = '}'$ 
end;
writeln ('CLAVES Y NUMEROS DE VECES QUE APARECEN:');
suma := 0; sumb := b[0];
for  $i := 1$  to  $n$  do
begin suma := suma + a[i]; sumb := sumb + b[i];
writeln(b[i - 1], a[i], ' ', clave[i])
end;

```

Programa 4.6. (Continuación)

```

writeln(b[n]);
writeln(' _____ ');
writeln(sumb, suma);
{calcular p a partir de a y b}
for i := 0 to n do
begin p[i, i] := b[i];
  for j := i + 1 to n do p[i, j] := p[i, j - 1] + a[j] + b[j]
end;
write ('LONGITUD DE CAMINO MEDIA DEL ARBOL EQUILIBRADO = ');
writeln(arbolequi(0, n)/p[0, n] : 6 : 3); imprimirarbol;
arbolopt;
writeln('LONGITUD DE CAMINO MEDIA DEL ARBOL OPTIMO = ');
writeln(c[0, n]/p[0, n] : 6 : 3); imprimirarbol;
{ahora se consideran solo las claves, haciendo b = 0}
for i := 0 to n do
begin p[i, i] := 0;
  for j := i + 1 to n do p[i, j] := p[i, j - 1] + a[j]
end;
arbolopt;
writeln('ARBOL OPTIMO CONSIDERANDO SOLO LAS CLAVES ');
imprimirarbol
end .

```

Programa 4.6. (Continuación)

4.5. ARBOLES MULTICAMINO

Hasta ahora, el análisis se ha limitado a los árboles en que cada nodo tiene como máximo dos descendientes, es decir, a los árboles binarios. Esto resulta perfectamente apropiado si, por ejemplo, se quieren representar relaciones familiares en las que cada persona se encuentre relacionada con sus padres. Despu^s de todo, nadie tiene más de dos progenitores! Pero, ¿qué ocurre si se prefiere ver la relación en sentido inverso? Es preciso entonces tener en cuenta el hecho de que algunas personas tienen más de dos hijos y, consecuentemente, los árboles que representen la relación familiar contendrán nodos con muchas ramas. A falta de un término mejor, se denominará a estas estructuras *árboles «multicamino»*.

Desde luego, no existe nada realmente especial en tales estructuras, y ya no han tratado todos los mecanismos de programación y de definición de datos necesarios para manejarlas. Si, por ejemplo, se tiene un límite superior del número de hijos (lo cual es, desde luego, un supuesto bastante futurista), entonces no pueden representar los hijos como un componente, tipo array, del registro que

representa una persona. Sin embargo, si el número de hijos varía mucho de unas personas a otras, se puede tener una utilización de la memoria muy mala. En este caso, será mucho más apropiado organizar los hijos como una lista lineal, asignando a los padres un puntero al hijo más joven (o mayor). En (4.80) se da la definición de un tipo que podría servir en este caso y en la Fig. 4.43 se muestra una posible estructura de datos.

```

type persona = record nombre: alfa;
  hermano: ↑persona;
  hijo: ↑persona
end

```

(4.80)

Se observa que, si se inclina esta figura 45°, tiene la forma de un árbol binario perfecto. Pero este punto de vista induce a confusión porque, funcionalmente, los dos punteros tienen significados completamente distintos. Habitualmente no puede tratarse impunemente a un hermano igual que a un hijo y, por lo tanto, ni siquiera debe hacerse cuando se construyen definiciones de datos. Se podría convertir este ejemplo en una estructura de datos todavía más complicada, introduciendo más componentes en el registro de cada persona y representando con ellos otras relaciones familiares. Una de ellas, que no puede ser deducida, en el caso general, de las relaciones hijo y hermano, es la de marido y mujer; otra, la relación inversa de padre y madre. Tales estructuras crecen rápidamente, y pasan a ser complejos «bancos de datos relacionales», pudiéndose incluso representar varios árboles con una única estructura. Los algoritmos que trabajan con estas organizaciones de datos están íntimamente ligados a las definiciones de los datos y, por lo tanto, no tiene sentido especificar técnicas de aplicación general.

Sin embargo, hay un área muy práctica en la que se utilizan árboles multicamino, que sí es de interés general. Esta es la construcción y el mantenimiento de árboles de búsqueda en gran escala, dónde hay que hacer inserciones y borrados de elementos, pero en los que la memoria principal es demasiado costosa, o no suficientemente grande, para ser utilizada como almacenamiento permanente.

Supóngase, entonces, que los nodos de un árbol deben ser guardados en un dispositivo de almacenamiento secundario como, por ejemplo, un disco. Las estructuras de datos dinámicas presentadas en este capítulo son especialmente apropiadas para trabajar con dispositivos de memoria secundaria. La única novedad es el hecho de que los punteros se representan por direcciones del disco, en vez de direcciones de la memoria principal. Si se utiliza un árbol binario para un conjunto de datos que tenga, por ejemplo, un millón de elementos, se necesitarán $\log_2 10^6 \approx 20$ pasos de búsqueda como media. Como ahora cada paso necesita un acceso al disco (con su correspondiente tiempo de «latencia»), será muy deseable poder tener una organización de los datos que requiera un menor número de accesos al disco. El árbol multicamino es una solución perfecta para

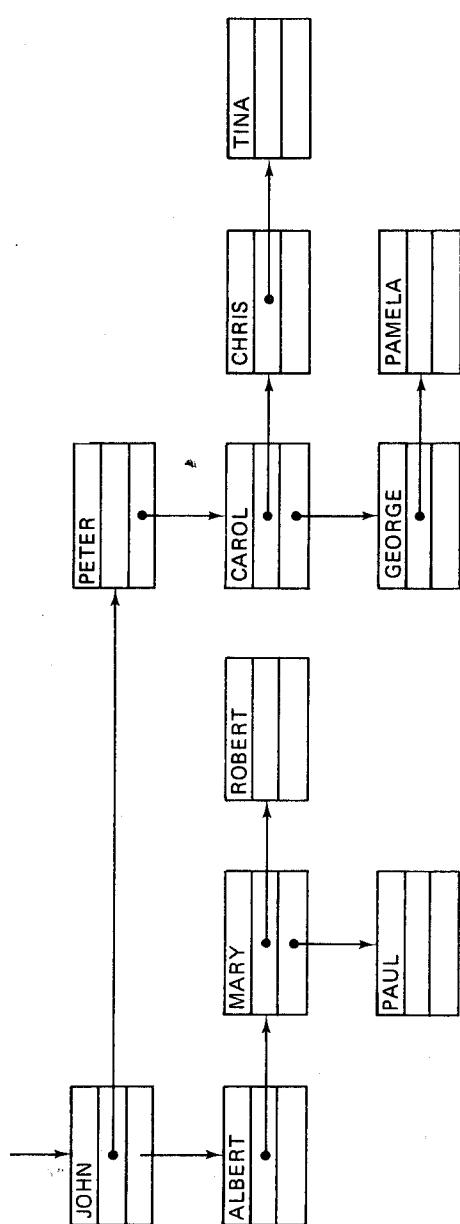


Fig. 4.43. Un árbol multicamino.

este problema. Si se accede a un elemento que está almacenado en memoria secundaria, también se puede acceder a un grupo completo de elementos sin que sea preciso mucho esfuerzo adicional. Esto sugiere dividir un árbol en subárboles y representar los subárboles como unidades a las que se accede en bloque. Estos subárboles se llamarán *páginas*. La Fig. 4.44 presenta un árbol binario subdividido en páginas.

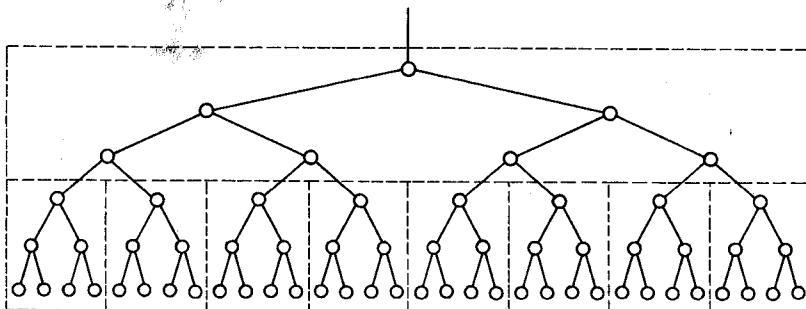


Fig. 4.44. Un árbol binario subdividido en «páginas».

El ahorro en el número de accesos al disco —cada acceso a una página representa ahora un acceso al disco— puede ser considerable. Supóngase que se decide colocar 100 nodos en cada página (este es un número razonable); entonces la búsqueda en el millón de elementos requerirá como media sólo $\log_{100} 10^6 = 3$ accesos de página en vez de 20. Desde luego, si se deja crecer el árbol «aleatoriamente», el caso más desfavorable puede necesitar 10^4 accesos! Resulta evidente que, en el caso de árboles multicamino, es imperativo que exista un método que controle el crecimiento del árbol.

4.5.1. Árboles B

Al buscar una forma de controlar el crecimiento, el árbol perfectamente equilibrado se desecha inmediatamente debido al gran esfuerzo que requiere la operación de reequilibrado. Hay que relajar un poco las reglas. R. Bayer [4.2], en 1970, propuso un criterio muy razonable: todas las páginas (excepto una) contienen entre n y $2n$ nodos, siendo n una constante dada. Por lo tanto, para un árbol de N elementos y tamaño de página máximo de $2n$ nodos, el caso más desfavorable requiere $\log_n N$ accesos de página —y está claro que los accesos de página son la parte más importante del esfuerzo de búsqueda—. Además, el factor de utilización de la memoria, punto importante, es por lo menos del 50 %, ya que las páginas están ocupadas como mínimo hasta su mitad. Además de todas estas ventajas, este método solo necesita algoritmos relativamente sencillos para realizar la búsqueda, inserción, y borrado de elementos. A continuación se estudian estos algoritmos en detalle.

Las estructuras de datos subyacentes se llaman árboles B y tienen las siguientes características (n se llama el *orden* del árbol B):

1. Cada página contiene como máximo $2n$ elementos (claves).
2. Cada página, excepto la página raíz, contiene al menos n elementos.
3. Cada página es, bien una página hoja, es decir, no tiene descendientes, o tiene $m + 1$ descendientes, siendo m el número de claves que tiene.
4. Todas las páginas hoja se encuentran en el mismo nivel.

La Fig. 4.45 muestra un árbol B de orden 2 con 3 niveles. Todas las páginas contienen 2, 3 ó 4 elementos; la excepción es la raíz, que puede tener un único elemento. Todas las páginas hoja se encuentran en el nivel 3. Si se comprime el árbol B en un único nivel, a base de insertar los descendientes entre las claves de

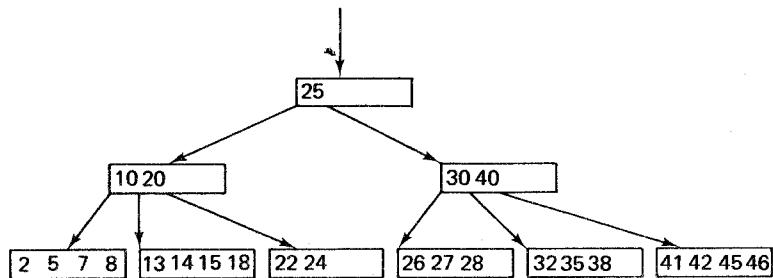


Fig. 4.45. Árbol B de orden 2.

su página antecesora, las claves resultan ordenadas de izquierda a derecha en orden creciente. Esta organización supone una extensión natural de la de los árboles binarios de búsqueda, y determina el método a utilizar para encontrar un elemento con clave dada. Considérese una página de la forma que se indica en la Fig. 4.46 y un argumento a buscar, x , dado. Suponiendo que se ha trasladado la página a la memoria principal, se pueden utilizar métodos de búsqueda convencionales entre las claves $k_1 \dots k_m$. Si m es suficientemente grande, se puede utilizar la búsqueda binaria; si m es bastante pequeña, una búsqueda secuencial ordinaria será suficiente. (Obsérvese que el tiempo necesario para realizar una búsqueda en memoria principal será probablemente insignificante, comparado con el tiempo necesario para mover la página de la memoria secundaria a la memoria principal). Si la búsqueda es infructuosa, se estará en una de las siguientes situaciones:

1. $k_i < x < k_{i+1}$, para $1 \leq i < m$. La búsqueda continúa en la página p_i .
2. $k_m < x$. La búsqueda continúa en la página p_m .
3. $x < k_1$. La búsqueda continúa en la página p_0 .

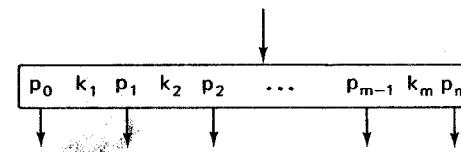


Fig. 4.46. Página de árbol B con m claves.

Si en algún caso el puntero es nil, es decir, si no hay página descendiente, entonces no hay ningún elemento en todo el árbol que tenga clave x , y se acaba la búsqueda.

Sorprendentemente, la inserción en un árbol B es también relativamente simple. Si hay que insertar un elemento en una página que tiene $m < 2n$ elementos, el proceso de inserción involucra a esa página únicamente. Sólo la inserción en una página llena tiene consecuencias que afectan a la estructura del árbol, y puede necesitar la creación de páginas nuevas. Para comprender lo que ocurre en este caso, véase la Fig. 4.47, que ilustra la inserción de la clave 22 en un árbol B de orden 2. La inserción se realiza en los pasos siguientes:

1. Se comprueba que la clave 22 no se encuentra en el árbol; la inserción en la página C no es posible porque C ya está completa.
2. La página C se *parte* en dos páginas (es decir, se crea una nueva página, D).
3. Las $m + 1$ claves se distribuyen equitativamente entre C y D, y la clave que está en el medio se sube un nivel a la página antecesora A.

Este esquema de funcionamiento es muy elegante y conserva todas las propiedades características de los árboles B. En particular, las páginas «partidas» tienen exactamente n nodos. Desde luego, la inserción de un elemento en la página antecesora puede causar desbordamiento de esa página, con el resultado de que el proceso de partición se propaga. En el caso extremo, puede propagarse hasta la raíz. Esta es, de hecho, la única forma en que un árbol B puede aumentar su altura. El árbol B tiene una forma extraña de crecer: lo hace desde sus hojas en dirección a la raíz.

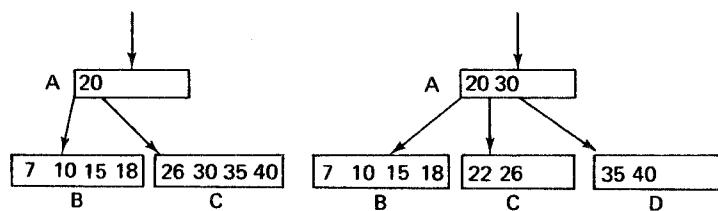


Fig. 4.47. Inserción de la clave 22 en un árbol B.

A continuación se desarrollará un programa detallado, partiendo de estas descripciones preliminares. Se ve de entrada que una formulación recursiva

va a ser la más conveniente, debido a que el proceso de división de los nodos se propaga en sentido inverso por el camino de búsqueda. La estructura general del programa será pues similar a la de la inserción en árboles equilibrados, aunque los detalles sean distintos.

Antes que nada, hay que formular una definición de la estructura de la página. Se decide representar los elementos en forma de array.

```
type pagina = record m: indice;
  p0: ref;
  e: array[1 .. nn] of item
end
```

(4.81)

siendo

```
const nn = 2 * n;
type ref = ↑pagina;
indice = 0 .. nn
```

y

```
type item = record clave: integer;
  p: ref;
  contador: integer
end
```

(4.82)

De nuevo el componente del ítem, *contador*, representa toda la información adicional que pudiera estar asociada con cada ítem, pero no juega ningún papel en el proceso de búsqueda en sí. Obsérvese que cada página tiene espacio para $2n$ ítems. El campo *m* indica cuántos de ellos posee la página en un momento determinado. Como $m \geq n$ (exceptuando la página raíz), se garantiza un uso de la memoria de, al menos, 50 %.

El algoritmo de búsqueda e inserción en un árbol B se formula en el procedimiento *buscar*, que forma parte del Programa 4.7. Su estructura principal es muy sencilla, y recuerda la de búsqueda en un árbol binario simple, excepción hecha de la decisión de bifurcación en cada nodo, que ahora no es una decisión entre dos opciones. En vez de ello, la «búsqueda interna a la página» se representa por una búsqueda binaria en el array *e*.

El algoritmo de inserción está formulado como procedimiento aparte, únicamente por razones de claridad del programa. Este resulta activado cuando *buscar* ha indicado que se debe mover un ítem hacia arriba en el árbol (en dirección a la raíz). Este hecho se indica por el parámetro boolean resultado *h*, que juega un papel muy similar al que tenía en el algoritmo de inserción en árbol equilibrado; *h* indica que el subárbol ha crecido. Si *h* es cierto, el segundo parámetro, *u*, representa el ítem que se está moviendo hacia arriba en el árbol.

Obsérvese que las inserciones comienzan en páginas hipotéticas, es decir en los «nodos especiales» de la Fig. 4.19; inmediatamente, el nuevo ítem se pasa, por medio del parámetro *u*, a la página hoja para ser verdaderamente insertado en ella. En (4.83) se muestra el proceso sin entrar en los detalles.

```
procedure buscar (x: integer; a: ref; var h: boolean; var u: item);
begin if a = nil then
  begin {x no esta en el arbol}
    Asignar x al ítem u y poner h a true para indicar que se envia
    un ítem u hacia arriba del arbol
  end else
  with a↑ do
    begin {buscar x en la pagina a↑}
      busqueda binaria en el array;
      if encontrado then
        incrementar el contador del ítem en cuestión else
      begin buscar(x, descendiente, h, u);
        if h then {se esta subiendo un ítem u}
          if (no. items en a↑) < 2n then
            insertar u en pagina a↑ y poner h a false
          else dividir pagina y subir el ítem que esta en el medio
        end
      end
    end
  end
end
```

(4.83)

Si el parámetro *h* tiene el valor true después de una llamada a *buscar* en el programa principal, hay que hacer una división de la página raíz. Esta operación debe programarse de forma separada, ya que la página raíz juega un papel especial. El proceso consiste simplemente en la creación de una nueva página (raíz), seguida de la inserción de un único ítem, dado por el parámetro *u*, en esa página. Como consecuencia, la nueva página raíz tiene un sólo ítem. Los detalles del proceso pueden verse en el Programa 4.7.

La Fig. 4.48 muestra el resultado de utilizar el Programa 4.7 para construir un árbol B, insertando la siguiente secuencia de claves:

```
20; 40 10 30 15; 35 7 26 18 22; 5; 42 13 46
27 8 32; 38 24 45 25;
```

Los punto y comas marcan los lugares en los que se han tomado las «instantáneas» de creación de nuevas páginas. La inserción de la última clave hace que se realicen dos divisiones de página y se creen tres nuevas de ellas.

Obsérvese el especial significado de la cláusula **with** en este programa. Aparece ya evidente en el esquema (4.83). En primer lugar, indica que los identificadores

de componentes de página se refieren, automáticamente, a la página $a \uparrow$, dentro de la instrucción prefijada por la cláusula. Si, de hecho, las páginas se encuentran en memoria secundaria —como sería necesario, ciertamente, en el sistema de un gran banco de datos— la cláusula **with** se puede interpretar adicionalmente como implicando el traslado a la memoria principal de la página en cuestión. Como cada activación de **buscar** implica entonces el movimiento a la memoria principal de una página, se necesitan como máximo $k = \log_2 N$ llamadas recursivas. Por lo tanto, si el árbol contiene N ítems, hay que poder almacenar k páginas en la memoria principal. Este es un factor que limita el tamaño de la página, $2n$. De hecho, se hace necesario almacenar más de k páginas, ya que la inserción puede obligar a dividir algunas de ellas. Una conclusión adicional es que conviene que la página raíz esté siempre en memoria, ya que todas las consultas al árbol arrancan de la raíz.

Otra ventaja de la organización árbol B es que se presta adecuada y económicamente, para actualización estrictamente secuencial del banco de datos. Todas las páginas se mueven a la memoria principal exactamente una sola vez.

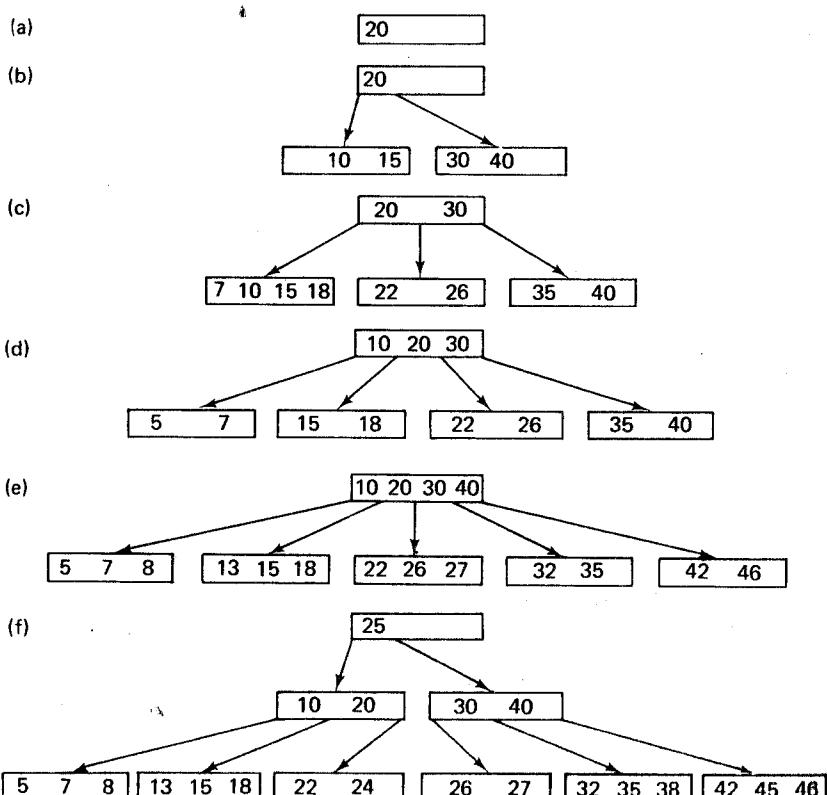


Fig. 4.48. Crecimiento de un árbol B de orden 2.

El *borrado* de ítems en un árbol B es un proceso bastante sencillo en principio, pero complicado en los detalles. Se distinguen dos casos:

1. El ítem a borrar se encuentra en una página hoja; entonces el algoritmo es claro y simple.
2. El ítem a borrar no se encuentra en una página hoja; debe sustituirse por uno de los dos ítems que le son lexicográficamente adyacentes, que resultan estar en páginas hoja y pueden ser borrados fácilmente.

En el caso 2 se encuentra la clave adyacente de una forma similar a como se hizo en el borrado en árbol binario. Se desciende a lo largo de los punteros más a la derecha hasta la página hoja P, se sustituye el ítem a borrar por el que está más a la derecha en P, y entonces se reduce el tamaño de P en una unidad.

En cualquier caso, la reducción del tamaño debe seguirse de una comprobación del número de ítems m en la página en cuestión, ya que si resulta $m < n$ dejará de cumplirse la característica fundamental de los árboles B. En ese caso debe hacerse una operación adicional que reorganice la estructura; esta *subocupación* se indica por medio del parámetro booleano variable, h .

La única alternativa es añadir un ítem tomado de una de las páginas vecinas. Como esta operación requiere traer la página Q a la memoria principal —operación relativamente costosa— se puede obtener el mayor provecho posible de esta situación desfavorable añadiendo más de un ítem de cada vez. Lo más frecuente es distribuir equitativamente, de una sola vez, los ítems de las páginas P y Q. Esto se llama *equilibrar*.

Desde luego, puede ocurrir que Q haya alcanzado ya su tamaño mínimo n y no haya ningún elemento que pueda tomarse para añadir a Q. En este caso, el número total de ítems en las páginas P y Q es $2n - 1$; se pueden *unir* las dos páginas en una, poniendo como ítem central uno de la página antecesora de P y Q, y eliminar completamente la página Q. Este es exactamente el proceso inverso de la división de una página. Este procedimiento se pone de relieve con la operación de borrado de la clave 22, en la Fig. 4.47.

Puede ocurrir, al sacar una clave de la página antecesora, que su tamaño baje también del nivel permitido n , necesitándose realizar más operaciones (equilibrar o unir) en el nivel siguiente. En el caso extremo, la unión de páginas puede propagarse hasta la raíz. Si la raíz queda reducida a un tamaño nulo se borra, produciéndose una reducción de la altura del árbol B. Esta es, de hecho, la única manera en que puede reducirse la altura de un árbol B.

La Fig. 4.49 muestra el proceso gradual de reducción del tamaño del árbol B de la Fig. 4.48 cuando se borran, en secuencia, las claves

25, 45, 24; 38, 32; 8, 27, 46, 13, 42; 5, 22, 18, 26; 7, 35, 15;

Como antes, los punto y comas marcan los lugares en los que se han tomado las «instantáneas», que son los puntos donde se eliminan páginas. El algoritmo de

borrado de ítems se incluye en el Programa 4.7 en forma de procedimiento. La analogía de su estructura con la del algoritmo de borrado en árbol equilibrado, es especialmente notable.

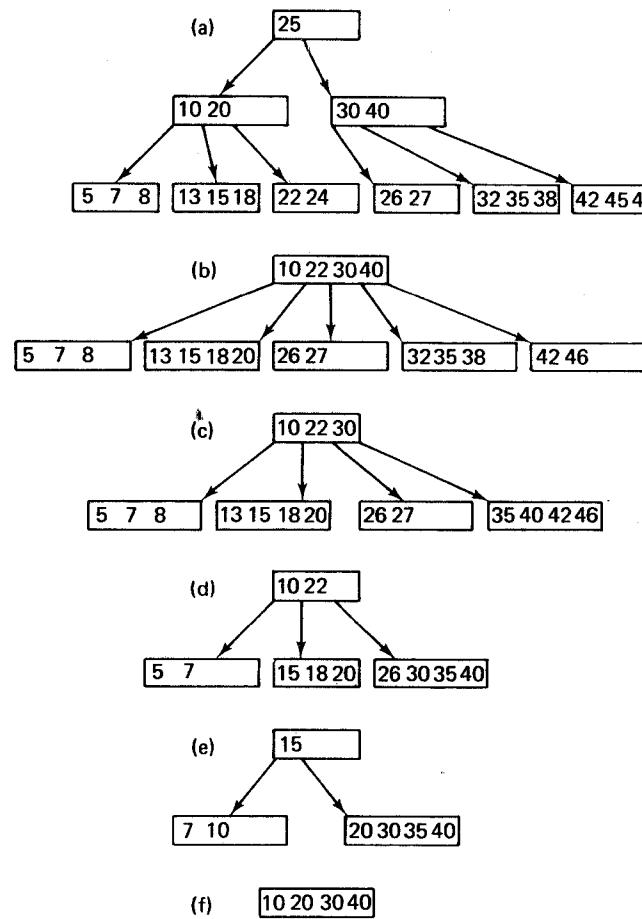


Fig. 4.49. Reducción de un árbol B de orden 2.

Bayer y McCreight han realizado un análisis exhaustivo del funcionamiento de los árboles B; los resultados obtenidos se encuentran en el artículo referenciado. En concreto, hay un estudio del tamaño de página, n , óptimo en donde se ve que éste depende mucho de las características de la memoria y el computador utilizados.

En Knuth, Vol. 3, páginas 476-479, se estudian variaciones del árbol B. El resultado más interesante es que se debe intentar equilibrar páginas colindantes antes de realizar la división de páginas, al igual que se ha hecho en el caso de la

Programa 4.7. Búsqueda, inserción y borrado en un árbol B.

```

program arbolB(input, output);
{busqueda, inserción y borrado en un arbol B}
const n = 2; nn = 4; {tamaño de página}
type ref = ^pagina;
item = record clave: integer;
p: ref;
contador: integer
end;
pagina = record m: 0 .. nn; {no. de items}
p0: ref;
e: array [1 .. nn] of item
end;
var raiz, q: ref; x: integer;
h: boolean; u: item;
procedure buscar(x: integer; a: ref; var h: boolean; var v: item);
{Buscar la clave x en el arbol B de raiz a; si se encuentra, se incrementa el contador; si no, se inserta en el arbol un item con clave x y contador a 1. Si debe moverse un item a un nivel mas bajo, se le asigna a v; h := «la altura del arbol a ha aumentado»}
var k, iz, de: integer; q: ref; u: item;
procedure insertar;
var i: integer; b: ref;
begin {insertar u a la derecha de a↑ . e[de]}
with a↑ do
begin if m < nn then
begin m := m + 1; h := false;
for i := m downto de + 2 do e[i] := e[i - 1];
e[de + 1] := u
end else
begin {la pagina a↑ esta llena; dividirla y asignar el item a mover a v}
new(b);
if de ≤ n then
begin if de = n then v := u else
begin v := e[n];
for i := n downto de + 2 do e[i] := e[i - 1];
e[de + 1] := u
end;
for i := 1 to n do b↑ . e[i] := a↑ . e[i + n]
end else
begin {insertar u en la pagina apropiada} de := de - n; v := e[n + 1];
for i := 1 to de - 1 do b↑ . e[i] := a↑ . e[i + n + 1];
b↑ . e[de] := u;
for i := de + 1 to n do b↑ . e[i] := a↑ . e[i + n]
end;
end;
end;
end;
end;

```

```

 $m := n; b \uparrow . m := n; b \uparrow . p0 := v . p; v . p := b$ 
end
end {with}
end {insertar};

begin {la clave x no esta en la pagina a\uparrow; h = falso}
if a = nil then
begin {el item con clave x no esta en el arbol} h := true;
with v do
begin clave := x; contador := 1; p := nil
end
end else
with a\uparrow do
begin iz := 1; de := m; {busqueda binaria en el array}
repeat k := (iz + de) div 2;
if x ≤ e[k].clave then de := k - 1;
if x ≥ e[k].clave then iz := k + 1;
until de < iz;
if iz - de > 1 then
begin {encontrado} e[k].contador := e[k].contador + 1; h := false
end else
begin {el item no esta en esta pagina}
if de = 0 then q := p0 else q := e[de].p;
buscar(x, q, h, u); if h then insertar
end
end
end {buscar};

procedure borrar(x: integer; a: ref; var h: boolean);
{buscar y borrar en un arbol B la clave x; si se produce subocupacion
de la pagina, equilibrar con la pagina adyacente si es posible, si no unir las dos
paginas; h := «la pagina a es demasiado pequena»}
var i, k, iz, de: integer; q: ref;
procedure subocupacion(c, a: ref; s: integer; var h: boolean);
{a = pagina subocupada, c = pagina antecesora}
var b: ref; i, k, mb, mc: integer;
begin mc := c\uparrow . m; {h = true, a\uparrow . m = n - 1}
if s < mc then
begin {b := pagina a la derecha de a} s := s + 1;
b := c\uparrow . e[s].p; mb := b\uparrow . m; k := (mb - n + 1) div 2;
{k := no. de items de la pagina adyacente b}
a\uparrow . e[n] := c\uparrow . e[s]; a\uparrow . e[n].p := b\uparrow . p0;
if k > 0 then

```

Programa 4.7. (Continuación)

```

begin {mover k items de b a a}
for i := 1 to k - 1 do a\uparrow . e[i + n] := b\uparrow . e[i];
c\uparrow . e[s] := b\uparrow . e[k]; c\uparrow . e[s].p := b;
b\uparrow . p0 := b\uparrow . e[k].p; mb := mb - k;
for i := 1 to mb do b\uparrow . e[i] := b\uparrow . e[i + k];
b\uparrow . m := mb; a\uparrow . m := n - 1 + k; h := false
end else
begin {unir paginas a y b}
for i := 1 to n do a\uparrow . e[i + n] := b\uparrow . e[i];
for i := s to mc - 1 do c\uparrow . e[i] := c\uparrow . e[i + 1];
a\uparrow . m := nn; c\uparrow . m := mc - 1; {dispose(b)}
end
end else
begin {b := pagina a la izquierda de a}
if s = 1 then b := c\uparrow . p0 else b := c\uparrow . e[s - 1].p;
mb := b\uparrow . m + 1; k := (mb - n) div 2;
if k > 0 then
begin {mover k items de la pagina b a la pagina a}
for i := n - 1 downto 1 do a\uparrow . e[i + k] := a\uparrow . e[i];
a\uparrow . e[k] := c\uparrow . e[s]; a\uparrow . e[k].p := a\uparrow . p0; mb := mb - k;
for i := k - 1 downto 1 do a\uparrow . e[i] := b\uparrow . e[i + mb];
a\uparrow . p0 := b\uparrow . e[mb].p;
c\uparrow . e[s] := b\uparrow . e[mb]; c\uparrow . e[s].p := a;
b\uparrow . m := mb - 1; a\uparrow . m := n - 1 + k; h := false
end else
begin {unir paginas a y b}
b\uparrow . e[mb] := c\uparrow . e[s]; b\uparrow . e[mb].p := a\uparrow . p0;
for i := 1 to n - 1 do b\uparrow . e[i + mb] := a\uparrow . e[i];
b\uparrow . m := nn; c\uparrow . m := mc - 1; {dispose(a)}
end
end
end {subocupacion};

procedure bor(p: ref; var h: boolean);
var q: ref; {a y k son globales}
begin
with p\uparrow do
begin q := e[m].p;
if q ≠ nil then
begin bor(q, h); if h then subocupacion(p, q, m, h)
end else
begin p\uparrow . e[m].p := a\uparrow . e[k].p; a\uparrow . e[k] := p\uparrow . e[m];
m := m - 1; h := m < n
end
end

```

Programa 4.7. (Continuación)

```

    end
  end {bor};
begin {borrar}
  if a = nil then
    begin writeln(' LA CLAVE NO ESTA EN EL ARBOL '); h := false
  end else
    with a↑ do
      begin iz := 1; de := m; {busqueda binaria en el array}
        repeat k := (iz + de) div 2;
          if x ≤ e[k] . clave then de := k - 1;
          if x ≥ e[k] . clave then iz := k + 1;
        until iz > de;
        if de = 0 then q := p0 else q := e[de] . p;
        if iz - de > 1 then
          begin {encontrada, ahora se borra e[k]}
            if q = nil then
              begin {a es una pagina terminal} m := m - 1; h := m < n;
                for i := k to m do e[i] := e[i + 1];
              end else
                begin bor(q, h); if h then subocupacion(a, q, de, h)
                end
            end else
              begin borrar(x, q, h); if h then subocupacion(a, q, de, h)
              end
          end
        end
      end {borrar};
procedure imprimirarbol(p: ref; niv: integer);
  var i: integer;
begin if p ≠ nil then
  with p↑ do
    begin for i := 1 to niv do write(' ');
      for i := 1 to m do write(e[i] . clave: 4);
      writeln;
      imprimirarbol(p0, niv + 1);
      for i := 1 to m do imprimirarbol(e[i] . p, niv + 1)
    end
  end;
begin raiz := nil; read(x);
  while x ≠ 0 do
    begin writeln(' CLAVE A BUSCAR ', x);
      buscar(x, raiz, h, u);
      if h then

```

Programa 4.7. (Continuación)

```

begin {insertar nueva pagina base} q := raiz; new(raiz);
  with raiz↑ do
    begin m := 1; p0 := q; e[1] := u
    end
  end;
  imprimirarbol(raiz, 1); read(x)
end;
read(x);
while x ≠ 0 do
  begin writeln(' BORRAR CLAVE ', x);
    borrar(x, raiz, h);
    if h then
      begin {se ha reducido el tamaño de la pagina base}
        if raiz↑ . m = 0 then
          begin q := raiz; raiz := q↑ . p0; {dispose(q)}
        end
      end;
    imprimirarbol(raiz, 1); read(x)
  end
end .

```

Programa 4.7. (Continuación)

unión de páginas. Aparte de este resultado, las otras mejoras sugeridas parecen ser de escasa importancia.

4.5.2. Árboles B binarios

La clase de árboles B que parece ser la menos interesante es la formada por los árboles B de primer orden ($n = 1$). Pero a veces es interesante prestar atención incluso a estos casos. Está claro, sin embargo, que los árboles B de primer orden no son útiles para representar, en memoria secundaria, grandes conjuntos de datos indexados y ordenados, ya que el 50 % de las páginas, aproximadamente, tendrán un único ítem. Por lo tanto, no se considerará el uso de memoria secundaria, y se estudiará de nuevo el problema de árboles de búsqueda con *un nivel de memoria* solamente.

Un árbol B binario (árbol BB) está formado por nodos (páginas) de uno o dos ítems. Por lo tanto, cada página contiene dos o tres punteros a páginas descendientes; debido a ello esta estructura se llama también *árbol 2-3*. De acuerdo con la definición de árboles B, todas las páginas hoja están en el mismo nivel,

y todas las páginas interiores del árbol BB tienen dos o tres descendientes (incluyendo la raíz). Dado que se trabaja en memoria principal únicamente, hay que utilizar ésta de manera óptima, y parece poco apropiado representar los ítems de un nodo en forma de array. Una alternativa es utilizar asignación dinámica, y enlazada por punteros, de memoria; es decir, dentro de cada nodo hay una lista enlazada de 1 ó 2 ítems. Como cada nodo tiene tres descendientes como máximo, se pueden combinar los punteros a los descendientes y los punteros de la lista de ítems, de la forma indicada en la Fig. 4.50. De esta manera, el nodo del árbol B pierde su identidad real, y los ítems hacen el papel de nodos de un árbol binario normal. Es preciso, sin embargo, poder distinguir entre punteros a descendientes

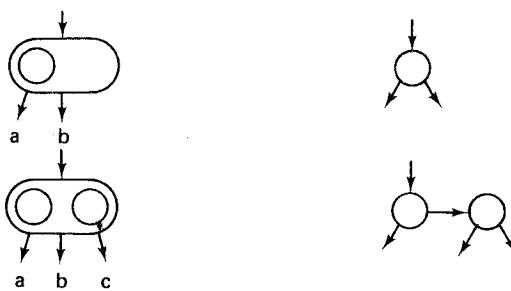


Fig. 4.50. Representación de los nodos de un árbol BB.

(verticales) y punteros a «hermanos» de la misma página (horizontales). Un único bit es suficiente para codificar esta información, ya que sólo los punteros derechos pueden ser horizontales. Se introduce por lo tanto el campo Booleano *h* que significa «horizontal». En (4.84) se da la definición de un nodo del árbol con esta representación. Esta fue sugerida y estudiada por R. Bayer en 1971 [4-3] y es una organización de árbol de búsqueda que garantiza una longitud máxima de camino $c = 2 \cdot \lceil \log N \rceil$.

```
type nodo = record clave: integer;
  .....
  izquierdo, derecho: ref;
  h: boolean
end
```

(4.84)

Al considerar el problema de la inserción de claves, hay que distinguir entre cuatro casos distintos, según crezca el subárbol derecho o el izquierdo. La Fig. 4.51 ilustra los cuatro casos. Recuérdese que los árboles B crecen desde abajo hacia la raíz y que todas las hojas deben permanecer al mismo nivel.

El caso más simple, (1), se presenta cuando el subárbol *derecho* de un nodo A crece, siendo A la única clave en su página (hipotética). Entonces, el descendiente B

se convierte, simplemente, en el hermano de A, es decir, el puntero vertical se transforma en puntero horizontal. Esta simple «elevación» del brazo derecho no es posible si A tiene ya un hermano, pues entonces se obtiene una página con tres nodos, y hay que dividirla (caso 2). Su nodo central B se sube al siguiente nivel superior.

Supóngase ahora que la altura del subárbol *izquierdo* de un nodo B ha crecido. Si otra vez B está solo en su página (caso 3), es decir, si su puntero derecho apunta a un descendiente, entonces A puede convertirse en el hermano de B. (Se necesita hacer una rotación de punteros simple, ya que el puntero izquierdo no puede ser horizontal.) Sin embargo, si B ya tiene un hermano, al subir A se obtiene una página con tres miembros, que hay que dividir. Se puede realizar esta división de una forma muy simple: se sube B al siguiente nivel superior y se transforma C en un descendiente de B (caso 4).

Debe observarse que, al buscar una clave dada, no existe diferencia entre avanzar por un puntero horizontal o uno vertical. Por lo tanto, parece artificioso preocuparse de si un puntero izquierdo en el caso 3 se ha convertido en horizontal, cuando su página todavía no tiene más de dos elementos. De hecho, el algoritmo de inserción muestra que existe una simetría extraña en el crecimiento de los subárboles izquierdo y derecho; ésto da una imagen bastante artificiosa a la organización del árbol BB.

No hay forma de «demostrar» esta rareza de la organización; sin embargo, una sana intuición dice que hay algo «extraño» y que debería eliminarse esta simetría. Se llega así a la noción de *árbol B binario simétrico* (árbol BBS) que también ha sido investigado por Bayer [4.4] en 1972. Esta organización conduce, por término medio, a árboles de búsqueda un poco más eficientes, pero la inserción y el borrado de elementos resultan ser operaciones algo más complejas. Además, ahora se necesitan dos bits en cada nodo (variables booleanas *hi* y *hd*) para indicar la naturaleza de sus dos punteros.

Restringiendo otra vez el estudio detallado al caso de la inserción, hay que distinguir entre cuatro casos. Estos se ilustran en la Fig. 4.52, que hace resaltar la simetría obtenida. Obsérvese que, cuando crece un subíndice de un nodo A que no tiene hermano, basta con hacer que la raíz del subárbol sea el hermano de A. Este caso no necesita más explicaciones.

En todos los cuatro casos de la Fig. 4.52 se presenta un desbordamiento de página seguido de una división de la misma. Los casos están etiquetados según las direcciones de los punteros que enlazan los tres hermanos en las figuras centrales. En la columna de la izquierda se presenta la situación inicial; la columna central muestra cómo el nodo inferior ha subido al crecer su subárbol; las figuras de la columna a la derecha muestran el resultado de reorganizar los nodos (división de página).

Es conveniente olvidar la organización de página que ha conducido a esta estructura, ya que lo único que interesa es limitar la longitud de camino, a, como máximo, $2 \cdot \lceil \log N \rceil$. Para ello, sólo se necesita garantizar que no existan dos punteros horizontales consecutivos en ningún camino de búsqueda. Sin

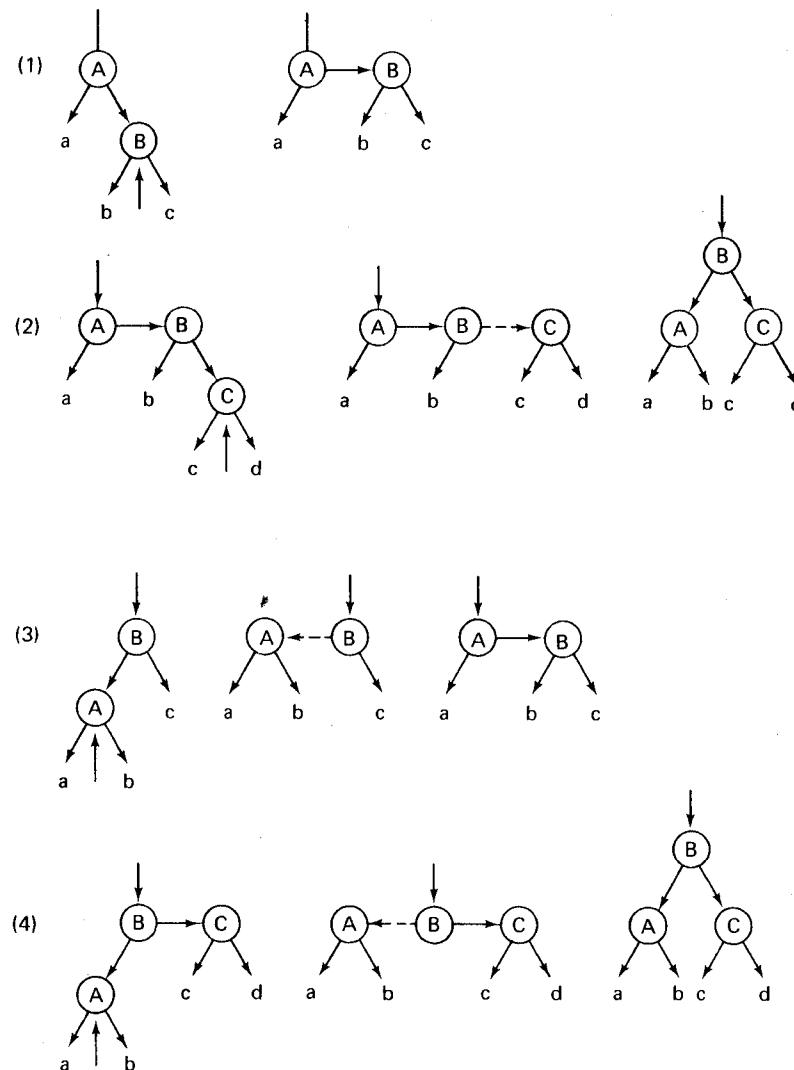


Fig. 4.51. Inserción de nodos en un árbol BB.

embargo, no existe razón para impedir que haya nodos con punteros horizontales a la derecha y a la izquierda. Se definirá por lo tanto el árbol BBS como un árbol que tiene las siguientes propiedades:

1. Todo nodo tiene una clave y un máximo de dos (punteros a) subárboles.

2. Todo puntero es horizontal o vertical. No hay dos punteros horizontales consecutivos en ningún camino de búsqueda.
3. Todos los nodos terminales (nodos sin descendientes) están en el mismo nivel (terminal).

A partir de esta definición se deduce que el camino de búsqueda más largo nunca es mayor que dos veces la altura del árbol. Como no hay árbol BBS de N nodos

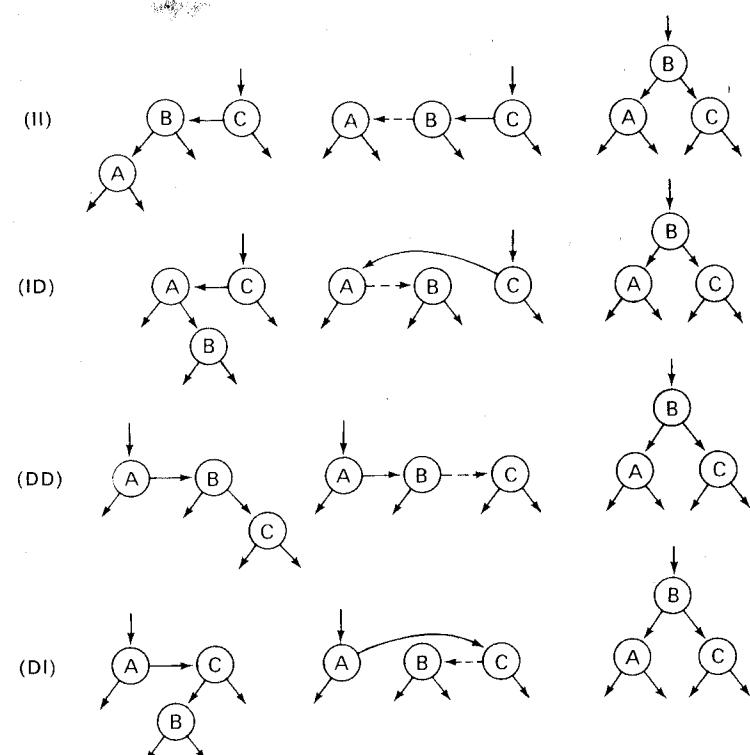


Fig. 4.52. Inserción en árboles BBS.

que tenga una altura mayor que $\lceil \log N \rceil$, se deduce inmediatamente que $2\lceil \log N \rceil$ es una cota superior de la longitud del camino de búsqueda.

Para poner de relieve la forma de crecer de estos árboles, obsérvese la Fig. 4.53. Cada fila de la figura representa el crecimiento del árbol al insertar las secuencias de claves siguientes (los punto y comas señalan los lugares donde se han tomado las «instantáneas»):

```
(1) 1 2; 3; 4 5 6; 7;
(2) 5 4; 3; 1 2 7 6;
(3) 6 2; 4; 1 7 3 5;
(4) 4 2 6; 1 7; 3 5;
```

(4.85)

Estas figuras muestran muy claramente la tercera propiedad de los árboles B: todos los nodos terminales aparecen en el mismo nivel. Por lo tanto, estas estructuras se parecen a setos de jardín recién podados. Se les llamará por ello *setos*.

El algoritmo de construcción de (árboles) setos se presenta en (4.87). Se basa en la definición de nodo (4.86), con los dos componentes *hi* y *hd* representando la horizontalidad de los punteros izquierdo y derecho.

```
type nodo = record clave: integer;
  contador: integer;
  izquierdo, derecho: ref;
  hi, hd: boolean
end
```

(4.86)

Nuevamente el procedimiento recursivo *buscar* sigue también el esquema del algoritmo básico de inserción en un árbol binario (ver 4.87). Se añade un tercer parámetro *h*; éste indica si el subárbol de raíz *p* ha cambiado o no, y se corresponde directamente con el parámetro *h* del programa de búsqueda en árbol B. Hay que distinguir entre el caso en que un subárbol (indicado por un puntero vertical) ha crecido, y el caso en que un nodo hermano (indicado por un puntero horizontal) ha obtenido otro hermano y es preciso, por consiguiente, dividir una página. El problema se resuelve fácilmente utilizando una *h* con tres valores, y los siguientes significados:

1. *h* = 0: el subárbol *p* no necesita cambios en la estructura del árbol.
2. *h* = 1: el nodo *p* ha obtenido un hermano.
3. *h* = 2: el subárbol *p* ha aumentado en altura.

```
procedure buscar(x: integer; var p: ref; var h: integer);
  var p1, p2: ref;
begin
  if p = nil then
    begin {la palabra no esta en el arbol; insertarla}
      new(p); h := 2;
      with p^ do
        begin clave := x; contador := 1; izquierdo := nil;
          derecho := nil; hi := false; hd := false
        end
    end
  end else
```

(4.87)

```
if x < p^.clave then
  begin buscar(x, p^.izquierdo, h);
    if h ≠ 0 then
      if p^.hi then
        begin p1 := p^.izquierdo; h := 2; p^.hi := false;
          if p1^.hi then
            begin {II} p1^.izquierdo := p1^.derecho;
              p1^.derecho := p; p1^.hi := false; p := p1
            end else
              if p1^.hd then
                begin {ID} p2 := p1^.derecho; p1^.hd := false;
                  p1^.derecho := p2^.izquierdo; p2^.izquierdo := p1;
                  p1^.izquierdo := p2^.derecho; p2^.derecho := p; p := p2
                end
              end else
                begin h := h - 1; if h ≠ 0 then p^.hi := true
                end
            end else
              if x > p^.clave then
                begin buscar(x, p^.derecho, h);
                  if h ≠ 0 then
                    if p^.hd then
                      begin p1 := p^.derecho; h := 2; p^.hd := false;
                        if p1^.hd then
                          begin {DD} p1^.derecho := p1^.izquierdo;
                            p1^.izquierdo := p; p1^.hd := false; p := p1
                          end else
                            if p1^.hi then
                              begin {DI} p2 := p1^.izquierdo; p1^.hi := false;
                                p1^.izquierdo := p2^.derecho; p2^.derecho := p1;
                                p1^.derecho := p2^.izquierdo; p2^.izquierdo := p; p := p2
                              end
                            end else
                              begin h := h - 1; if h ≠ 0 then p^.hd := true
                              end
                            end else
                              begin p^.contador := p^.contador + 1; h := 0
                              end
                end {buscar}
```

(4.87)

Obsérvese que las operaciones que se realizan al reorganizar la estructura recuerdan mucho las del algoritmo de búsqueda en un árbol equilibrado (4.63). Resulta evidente en (4.87) que todos los casos pueden ser realizados mediante rotaciones de punteros: rotaciones simples en los casos II y DD y rotaciones do-

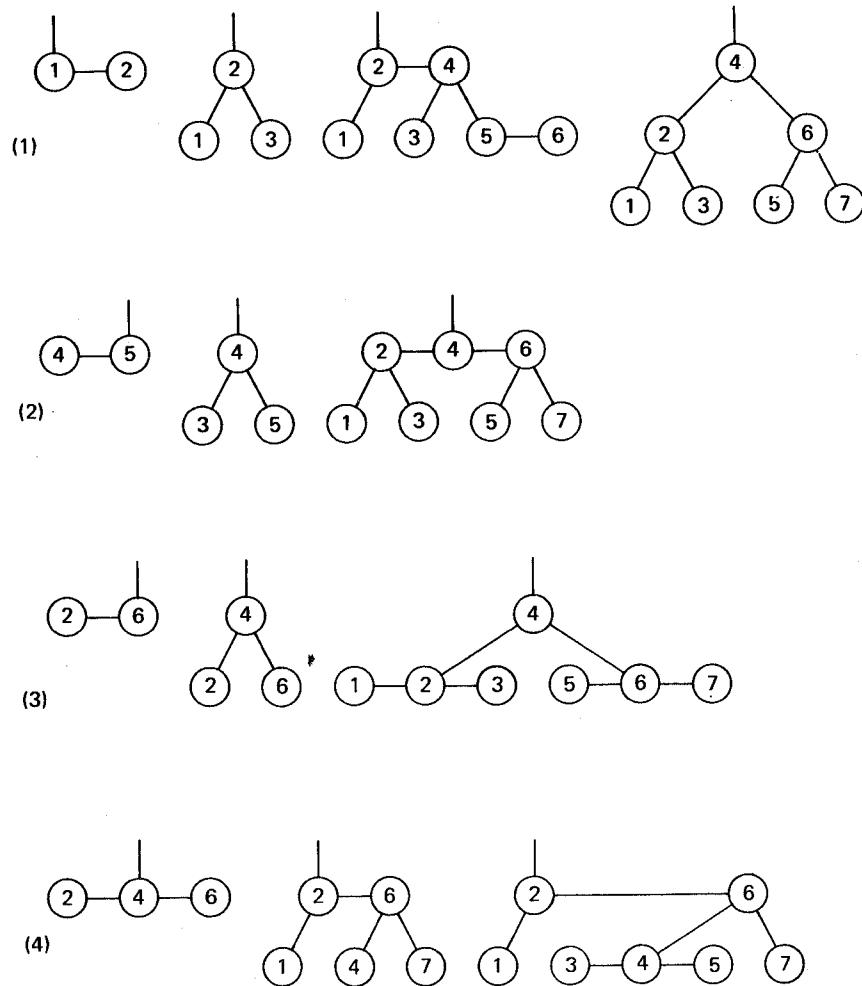


Fig. 4.53. Crecimiento de árboles «seto» al realizar las secuencias de inserciones (4.85).

bles en los casos *ID* y *DI*. De hecho, el procedimiento (4.87) es un poco más simple que el (4.63). Está claro que los árboles seto son una alternativa a los árboles AVL, y, por lo tanto, resulta deseable, y al mismo tiempo posible, establecer una comparación del funcionamiento de ambas estructuras.

Sin entrar en un análisis matemático detallado, se resaltarán algunas diferencias básicas. Se puede demostrar que los *árboles equilibrados AVL* son un subconjunto de los *árboles seto*. Es decir, la clase de estos últimos es más amplia. Por lo tanto, su longitud de camino será, por término medio, mayor que la de los

árboles AVL. En relación con esta observación, considérese el «caso más desfavorable» (4) de la Fig. 4.53. Por otro lado, los setos necesitan reorganizar los nodos menos frecuentemente. Consecuentemente, el árbol equilibrado es preferible en aplicaciones donde haya muchos más accesos a nodos que inserciones (o borrados); si este cociente es pequeño, la organización de árbol seto puede ser preferible.

Es muy difícil precisar mejor cuándo debe utilizarse una u otra organización. Depende mucho, no sólo del cociente entre el número de accesos y cambios de estructura, sino también de las características de la implantación. Esto es especialmente cierto si los registros de los nodos se representan de forma densamente empaquetada y, consecuentemente, el acceso a los distintos campos requiere seleccionar partes de una palabra. En muchas implantaciones, los campos Booleanos (*hi* y *hd* en el caso de los árboles seto) pueden ser manipulados mucho más fácilmente que los campos de tres valores (*equi* en el caso de árboles equilibrados).

4.6. TRANSFORMACIONES DE CLAVES («HASHING»)

En el apartado anterior se ha estudiado un problema que sirve para encontrar soluciones que utilizan técnicas de asignación dinámica de memoria. El problema es el siguiente:

Se tiene un conjunto *S* de elementos identificados por los valores de una clave, en los que está definida una relación de orden; ¿cómo debe organizarse *S* para que se pueda encontrar, con el mínimo esfuerzo, un elemento con valor de clave dado, *k*?

Evidentemente, en la memoria de un computador, cada elemento va a estar referenciado al final por una dirección *d* de la memoria. Por lo tanto, el problema expuesto puede entenderse como el de encontrar una *aplicación* adecuada *H* del conjunto *C* de las claves en el conjunto *D* de las direcciones:

$$H: C \rightarrow D$$

En el Apartado 4.5 se hizo esta *aplicación* con varios algoritmos de búsqueda en listas y árboles, basados en distintas organizaciones subyacentes de los datos. A continuación, se presenta otra solución más al mismo problema, que es básicamente muy sencilla y, en muchos casos, muy eficaz. Más adelante se analizarán algunas de las desventajas que tiene.

En esta solución se utiliza la estructura de datos array. Por lo tanto, *H* es una aplicación que transforma claves en índices del array, razón por la cual esta técnica se conoce usualmente con el nombre de *transformación de claves*. Hay que hacer notar que no se necesitarán mecanismos de asignación dinámica de memoria, ya que el array es una de las estructuras de datos fundamentales, estáticas.

Por lo tanto, este apartado está un poco fuera de lugar en el capítulo de estructuras dinámicas de información, pero conviene incluirlo aquí, ya que la utilización de la técnica de transformación de claves aparece en aplicaciones donde los árboles son otros posibles candidatos para la estructura a usar.

Cuando se transforman las claves, el problema principal es que el conjunto de claves posibles es mucho mayor que el conjunto de direcciones de memoria (índices del array) disponibles. Ejemplo típico es el uso de claves formadas por un máximo de diez letras para identificar miembros de un conjunto de, digamos, mil personas. Existen entonces 26^{10} claves distintas posibles que hay que relacionar con 10^3 índices. La función H es, por lo tanto, del tipo «muchos-a-uno». Dada una clave k en un proceso de búsqueda, el primer paso a dar es calcular su índice asociado $h = H(k)$, y el segundo —que es necesario, evidentemente— es comprobar si el elemento de clave k está realmente en el lugar h del array (tabla) T , es decir, comprobar si $T[H(k)].\text{clave} = k$. Dos preguntas surgen inmediatamente:

1. ¿Qué tipo de función H hay que utilizar?
2. ¿Qué se hace cuando H no da la posición del elemento deseado?

La respuesta a la segunda pregunta es que hay que usar algún método para obtener otra posición, con índice h' por ejemplo y, si ésta no es todavía la posición del elemento deseado, obtener un tercer índice h'' , y así sucesivamente. Cuando existe en la posición calculada una clave distinta de la deseada, se dice que ha ocurrido una *colisión*; el proceso de generar direcciones alternativas recibe el nombre de *manejo de colisiones*. A continuación se estudia cómo elegir la función de transformación y cómo manejar las colisiones.

4.6.1. Elección de la función de transformación

Es muy deseable que una buena función de transformación distribuya las claves en el campo de los valores índice de forma tan uniforme como sea posible. Aparte de esto, la función puede distribuir las claves de cualquier manera y, de hecho, es mejor si se comporta de forma completamente aleatoria. Esta propiedad ha hecho que el método se conozca con el nombre, no muy científico, «hashing»*, es decir, «trocear el argumento» o «desordenar», y H con el nombre *función hash*. Está claro que la función debe poder ser calculada eficientemente, es decir, debe estar formada por unas pocas operaciones aritméticas simples.

Supóngase que se dispone de una función $ord(k)$ que produce el número de orden de la clave k en el conjunto de todas las claves posibles. Supóngase, adicionalmente, que los índices i del array están comprendidos en el campo de enteros

* N. del T. No se ha podido encontrar un término adecuado en castellano para traducir esta palabra. Su significado se incluye a continuación en el texto.

$0 \dots N - 1$, siendo N la dimensión del array. En este caso, se tiene una solución obvia, que es la función:

$$H(k) = ord(k) \bmod N \quad (4.88)$$

Esta función tiene la propiedad de que distribuye uniformemente las claves en el campo de los índices y es, por lo tanto, la base de la mayor parte de las transformaciones de claves. Por otra parte, se puede calcular muy eficientemente si N es una potencia de 2. Pero esto es precisamente lo que hay que evitar cuando las claves son alfabéticas. En este caso, la hipótesis de que todas las claves son igualmente probables es completamente errónea. De hecho, es casi seguro que, claves que difieren en sólo algunas letras, sean asignadas el mismo índice con esta función de transformación, produciéndose una distribución muy poco uniforme. Por ello, es muy importante que, en (4.88), N sea un *número primo* [4-7]. Entonces no basta una mera truncación de dígitos binarios, sino que hay que hacer una división completa; esto no es un problema grave, sin embargo, ya que la mayor parte de los computadores modernos tienen la división como una de sus operaciones primitivas.

A menudo se utilizan funciones hash que están formadas por la aplicación de operaciones lógicas, tales como el «o exclusivo» a algunas partes de la clave, representada ésta como una secuencia de dígitos binarios. Estas operaciones lógicas son en algunas máquinas más rápidas que la división pero, a veces, fallan estrepitosamente, al producir una distribución de las claves, sobre el campo de índices, muy poco uniforme. Por esta razón, no se estudiarán aquí estos métodos en detalle.

4.6.2. Manejo de colisiones

Si sucede que el lugar de la tabla correspondiente a una cierta clave no contiene el elemento deseado, se dice que hay una colisión, es decir, que hay dos claves que corresponden al mismo índice. Se necesita entonces hacer un segundo intento en base a un índice secundario obtenido, a partir de la clave deseada, por un método determinista. Hay varios métodos para generar índices secundarios. Uno que es obvio, y al mismo tiempo eficaz, es enlazar todos los elementos que tienen el mismo índice primario $H(k)$, en forma de lista. Este método se llama *encadenamiento directo*. Los elementos de esta lista pueden estar en la tabla primaria o no. En el segundo caso, se suele llamar *área de desbordamiento* a la memoria donde están almacenados los elementos. Este método es muy eficiente, aunque tiene la desventaja de que necesita manipular listas secundarias, y tener espacio adicional, en cada elemento, para un puntero (o índice) a su lista de elementos colisionados.

Una solución alternativa al problema de las colisiones es eliminar todos los enlaces, y buscar simplemente en otros lugares de la tabla, hasta que se encuentre el elemento, o se llegue a un lugar vacío, en cuyo caso se puede suponer que la

clave especificada no se encuentra todavía en la tabla. Este método recibe el nombre de *direccionamiento vacío* [4-9]. Por supuesto, la secuencia de índices secundarios debe ser, para cada clave, siempre la misma. Se puede esquematizar el algoritmo de búsqueda en la forma:

```

 $h := H(k); i := 0;$ 
repeat
  if  $T[h] . clave = k$  then encontrado el elemento else
    if  $T[h] . clave = vacio$  then el elemento no esta en la tabla else (4.89)
    begin {colisión}
       $i := i + 1; h := H(k) + G(i)$ 
    end
  until encontrado o no esta en la tabla (o tabla llena)
```

Para resolver las colisiones, se han propuesto diversas funciones. Un compendio sobre el tema, publicado por Morris en 1968 [4-8], fue el origen de muchos trabajos en este campo. El método más simple es mirar el lugar siguiente—considerando la tabla organizada en forma circular—hasta encontrar el elemento buscado o llegar a una posición vacía. Esto significa hacer $G(i) = i$; los índices h_i , en este caso, son:

$$\begin{aligned} h_0 &= H(k) \\ h_i &= (h_0 + i) \bmod N, \quad i = 1 \dots N - 1 \end{aligned} \quad (4.90)$$

Este método se llama *inspección lineal* y tiene la desventaja de que los elementos tienden a *agruparse* alrededor de las claves primarias (claves que han sido insertadas sin colisionar). Idealmente, desde luego, debería escogerse una función G que, a su vez, distribuyera uniformemente las claves en los sitios restantes. En la práctica, sin embargo, esto tiende a ser demasiado costoso, y son preferibles métodos que sean sencillos de cálculo y superiores a la función lineal (4.90). Uno de éstos utiliza una función cuadrática que da la secuencia de índices siguiente:

$$\begin{aligned} h_0 &= H(k) \\ h_i &= (h_0 + i^2) \bmod N \quad (i > 0) \end{aligned} \quad (4.91)$$

Obsérvese que no es necesario elevar al cuadrado si se utilizan, para calcular el siguiente índice, las relaciones recurrentes (4.92), siendo $h_i = i^2$, $d_i = 2i + 1$, $h_0 = 0$ y $d_0 = 1$.

$$\begin{aligned} h_{i+1} &= h_i + d_i \quad (i > 0) \\ d_{i+1} &= d_i + 2 \end{aligned} \quad (4.92)$$

Este método se llama *inspección cuadrática* y, aunque prácticamente no necesita cálculo adicional, evita bien el agrupamiento primario. Una ligera desventaja es que el método no inspecciona todos los lugares de la tabla, es decir, puede no encontrarse dónde insertar un nuevo elemento, a pesar de haber lugares vacíos. Si N es un *número primo*, la *inspección cuadrática* recorre, de todas formas, al menos la *mitad* de la tabla. Esto puede comprobarse observando que, si las inspecciones i y j coinciden en el mismo lugar de la tabla, se cumple que

$$i^2 \bmod N = j^2 \bmod N$$

o que

$$(i^2 - j^2) \equiv 0 \pmod{N}$$

Descomponiendo en dos factores el primer término se obtiene:

$$(i + j)(i - j) \equiv 0 \pmod{N}$$

Como $i \neq j$, tiene que ocurrir que i o j sean *al menos* $N/2$ para obtener $i + j = cN$, siendo c un entero.

En la práctica, la desventaja de este método no tiene importancia, ya que es muy raro que se presenten $N/2$ colisiones consecutivas, y sólo sucede cuando la tabla está casi llena.

Como aplicación de las técnicas de «memoria dispersa» el generador de referencias cruzadas del Programa 4.5 aparece redactado de nuevo como Programa 4.8. Las diferencias principales entre ambos están en el procedimiento *buscar* y el cambio del tipo puntero *refpalabra* por la tabla de palabras T . La función hash H es la función módulo n siendo n el tamaño de la tabla; se ha elegido manejar las colisiones mediante inspección cuadrática. Obsérvese que, para que el proceso tenga un buen rendimiento, es fundamental que el tamaño de la tabla sea un número primo.

Aunque el método de transformación de claves es el más eficaz en este caso —es de hecho más eficaz que las organizaciones árbol—, también tiene una desventaja. Después de examinar el texto y almacenar las palabras, se desea imprimirlas en orden alfabetico. Esto es muy fácil de hacer cuando se utiliza una organización árbol, ya que el método está basado en el árbol de búsqueda ordenado. No es tan fácil de hacer, sin embargo, cuando se utiliza el método de transformación de claves. Es aquí cuando aparece claro el significado del término *hashing*. No sólo se necesita ordenar las palabras antes de imprimirlas (por razones de sencillez el Programa 4.8 hace la ordenación por el método de inserción directa), sino que resulta conveniente incluso mantener una lista enlazada especial con todas las claves que se han insertado. Por lo tanto, la mejora obtenida con el método hashing en el acceso a los elementos, se ve parcialmente obscurecida por

las operaciones adicionales necesarias para realizar la tarea completa de generación de un índice ordenado de referencias cruzadas.

Programa 4.8. Generador de referencias cruzadas utilizando una tabla hash.

```

program refcruzadas(f, output);
{generador de referencias cruzadas utilizando una tabla hash}
label 13;
const c1 = 10; {longitud de las palabras}
c2 = 8; {numeros por linea}
c3 = 6; {digitos por numero}
c4 = 9999; {numero de linea maximo}
p = 997; {numero primo}
libre = ' ';
type indice = 0 .. p;
refitem = ^item;
palabra = record clave: alfa;
            primero, ultimo: refitem;
            suce: indice
          end;
item = packed record
            nol: 0 .. c4;
            sig: refitem
          end;
var i, sup: indice;
k, k1: integer;
n: integer; {numero de linea en curso}
id: alfa;
f: text;
a: array [1 .. c1] of char;
t: array [0 .. p] of palabra; {tabla hash}
procedure buscar;
var h, d, i: indice;
x: refitem; e: boolean;
{variables globales: t, id, sup}
begin h := ord(id) mod p;
e := false; d := 1;
new(x); x^.nol := n; x^.sig := nil;
repeat
  if t[h].clave = id then
    begin {encontrado} e := true;
      t[h].ultimo^.sig := x; t[h].ultimo := x
    end else

```

```

if t[h].clave = libre then
begin {elemento nuevo} e := true;
  with t[h] do
    begin clave := id; primero := x; ultimo := x; suce := sup
    end;
    sup := h
end else
begin {colision} h := h + d; d := d + 2;
  if h ≥ p then h := h - p;
  if d = p then
    begin writeln('DESBORDAMIENTO DE LA TABLA'); goto 13
    end
  end
until e
end {buscar};
procedure imprimirtabla;
var i, j, m: indice;
procedure imprimirpalabra(p: palabra);
var l: integer; x: refitem;
begin write(' ', p^.clave);
  x := p^.primero; l := 0;
repeat if l = c2 then
  begin writeln;
    l := 0; write(' ': c1 + 1)
  end;
  l := l + 1; write(x^.nol: c3); x := x^.sig
until x = nil;
writeln
end {imprimirpalabra};
begin i := sup;
while i ≠ p do
begin {examinar la lista enlazada y encontrar la clave minima}
  m := i; j := t[i].sucess;
  while j ≠ p do
    begin if t[j].clave < t[m].clave then m := j;
      j := t[j].sucess
    end;
  imprimirpalabra(t[m]);
  if m ≠ i then
    begin t[m].clave := t[i].clave;
      t[m].primero := t[i].primero; t[m].ultimo := t[i].ultimo
    end;

```

Programa 4.8. (Continuación)

```

i := t[i].suce
end
end {imprimirtbla};
begin n := 0; k1 := c1; sup := p; reset(f);
  for i := 0 to p do t[i].clave := libre;
  while  $\neg \text{eof}(f)$  do
    begin if n = c4 then n := 0;
      n := n + 1; write(n: c3); {linea siguiente}
      write(' ');
      while  $\neg \text{eoln}(f)$  do
        begin {examinar linea}
          if f↑ in ['A' .. 'Z'] then
            begin k := 0;
            repeat if k < c1 then
              begin k := k + 1; a[k] := f↑;
              end;
              write(f↑); get(f)
            until  $\neg (f↑ \text{ in } ['A' .. 'Z', '0' .. '9'])$ ;
            if k ≥ k1 then k1 := k else
              repeat a[k1] := ' '; k1 := k1 - 1
            until k1 = k;
            pack(a, 1, id); buscar;
          end else
          begin {ver si es comilla o comentario}
            if f↑ = " " then
              repeat write(f↑); get(f)
              until f↑ = " " else
            if f↑ = '{ then
              repeat write(f↑); get(f)
              until f↑ = '}';
              write(f↑); get(f)
            end
          end;
          writeln; get(f)
        end;
      13: page; imprimirtbla
    end.

```

Programa 4.8. (Continuación)

4.6.3. Análisis del Método de transformación de claves

Evidentemente, la transformación de claves tiene, en el caso más desfavorable, un rendimiento pésimo en la inserción y recuperación de ítems. Después de todo, es perfectamente posible que la clave a buscar sea tal, que todos los intentos

de encontrarla den consistentemente con lugares llenos sin encontrar el elemento deseado (o un lugar vacío). De hecho, para utilizar la técnica hash, se necesita tener mucha confianza en la veracidad de las leyes de la teoría de probabilidades. Hace falta saber que, por *termino medio*, el número de pasos necesarios, para insertar/recuperar una clave, es pequeño. A continuación se hace un razonamiento probabilístico que revela que este número es, no sólo pequeño, sino *muy pequeño*.

Supóngase nuevamente que todas las claves son igualmente probables y la función hash H las distribuye uniformemente en el campo de índices de la tabla. Supóngase, además, que hay que insertar una clave en una tabla, de tamaño n , que ya tiene k elementos. La probabilidad de alcanzar un lugar vacío al primer intento es $1 - k/n$. Esta es también la probabilidad, p_1 , de que se necesite una única comparación. La probabilidad de que sea necesario un segundo intento exactamente es igual a la probabilidad de que se haya producido una colisión al primer intento, multiplicada por la probabilidad de que se acierte en un lugar vacío la segunda vez. En general, la probabilidad p_i de que una inserción requiera i intentos exactamente es:

$$\begin{aligned}
 p_1 &= \frac{n-k}{n} \\
 p_2 &= \frac{k}{n} \cdot \frac{n-k}{n-1} \\
 p_3 &= \frac{k}{n} \cdot \frac{k-1}{n-1} \cdot \frac{n-k}{n-2} \\
 &\vdots \\
 p_i &= \frac{k}{n} \cdot \frac{k-1}{n-1} \cdot \frac{k-2}{n-2} \cdot \dots \cdot \frac{k-i+2}{n-i+2} \cdot \frac{n-k}{n-i+1}
 \end{aligned} \tag{4.93}$$

Por lo tanto, el número de intentos esperado, para insertar la clave $(k+1)$ -ésima es:

$$\begin{aligned}
 E_{k+1} &= \sum_{i=1}^{k+1} i \cdot p_i = 1 \cdot \frac{n-k}{n} + 2 \cdot \frac{k}{n} \frac{n-k}{n-1} + \dots + (k+1) \cdot \\
 &\quad \cdot \left(\frac{k}{n} \frac{k-1}{n-1} \frac{k-2}{n-2} \dots \frac{1}{n-k+1} \right) = \frac{n+1}{n-k+1}
 \end{aligned} \tag{4.94}$$

Como el número de intentos necesario para insertar un elemento es el mismo que el que se necesita para recuperarlo, (4.94) sirve para calcular el número medio de intentos, E , necesarios para encontrar una clave cualquiera (aleatoria) en una tabla. Sea n la dimensión de la tabla y m el número de claves que hay en ella. Se tiene:

$$E = \frac{1}{m} \sum_{k=1}^m E_k = \frac{n+1}{m} \sum_{k=1}^m \frac{1}{n-k+2} = \frac{n+1}{m} (H_{n+1} - H_{n-m+1}) \tag{4.95}$$

siendo

$$H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$$

la función armónica. H_n puede aproximarse como $H_n \approx \ln(n) + \gamma$, siendo γ la constante de Euler. Si se hace además la sustitución $\alpha = m/(n+1)$, se obtiene:

$$\begin{aligned} E &= \frac{1}{\alpha} (\ln(n+1) - \ln(n-m+1)) = \frac{1}{\alpha} \ln \frac{n+1}{n+1-m} \\ &= \frac{-1}{\alpha} \ln(1-\alpha) \end{aligned} \quad (4.96)$$

α es, aproximadamente, el cociente del número de lugares de la tabla que están ocupados, dividido por el número de lugares totales, y se llama el *factor de carga*; para una tabla vacía $\alpha = 0$ y para una tabla llena $\alpha = n/(n+1)$. El número de intentos esperado, E , para realizar una inserción o una recuperación, de una clave elegida aleatoriamente, aparece en la Tabla 4.6 como función del factor de carga α . Los resultados numéricos son en verdad sorprendentes, y explican el rendimiento, excepcionalmente bueno, del método de transformación de claves. Incluso cuando la tabla está ocupada en un 90 %, se puede encontrar una clave deseada o un lugar vacío con, por término medio, ¡sólo 2,56 intentos! En particular, obsérvese que esta cantidad no depende del número de claves que haya, sino del factor de carga únicamente.

α	E
0.1	1.05
0.25	1.15
0.5	1.39
0.75	1.85
0.9	2.56
0.95	3.15
0.99	4.66

Tabla 4.6. Número de intentos esperado en función del factor de carga.

Se ha basado el análisis anterior en la hipótesis de que el método de manejo de las colisiones distribuye las claves uniformemente en el campo de los lugares restantes. Los métodos utilizados en la práctica dan un rendimiento algo inferior al calculado. En (4.97) se da el número de intentos medio, obtenido al analizar detalladamente el método de *inspección lineal* [4-10].

$$E = \frac{1}{2} \frac{\alpha/2}{\alpha} \quad (4.97)$$

En la Tabla 4.7 se presentan algunos valores de $E(\alpha)$. Los resultados obtenidos con el método más simple de transformación de claves son tan buenos que podría pensarse que el método (hashing) es una panacea para todo.

α	E
0.1	1.06
0.25	1.17
0.5	1.50
0.75	2.50
0.9	5.50
0.95	10.50

Tabla 4.7. Número esperado de intentos con inspección lineal.

Esta creencia se debe especialmente a que su rendimiento es superior, incluso, a las organizaciones más sofisticadas de árboles vistos, al menos en lo que se refiere a recuperación e inserción. Resulta por ello importante señalar explícitamente algunas de las desventajas del hashing, incluso aunque éstas sean obvias si se hace una evaluación objetiva.

Desde luego, su mayor desventaja, en relación con las técnicas de asignación dinámica de memoria, es que *el tamaño de la tabla es fijo*, y no puede ajustarse según cambian las necesidades. Por lo tanto, hay que establecer una buena estimación del mismo *a priori*, si quiere evitarse una mala utilización de la memoria o un rendimiento pobre (o, incluso, un desbordamiento de la tabla). Incluso cuando se conoce exactamente el número de elementos —caso extremadamente raro— la tabla debe estar un poco (10 % por ejemplo) sobredimensionada para tener un buen rendimiento.

La segunda desventaja principal de las técnicas de memoria dispersa se hace patente si las claves no sólo van a ser insertadas y recuperadas, sino también borradas, ya que el *borrado* de elementos en una tabla hash es un proceso muy *pesado*, a menos que se utilice encadenamiento directo de elementos en un área de desbordamiento aparte. Es, por consiguiente, justo decir que las organizaciones árbol siguen siendo atractivas, y de hecho preferibles, cuando el volumen de los datos no se conoce bien, es muy variable, y, a veces, incluso disminuye.

E J E R C I C I O S

- 4.1. Supóngase que se introduce la noción de *tipo recursivo*,

$$\text{rectype } T = T_0$$

significando la unión del conjunto de valores definido por el tipo T_0 y el valor aislado *nada*, es decir,

$$T = T_0 \cup \{\text{nada}\}.$$

Entonces, la definición del tipo *arb*(ver (4.3)), por ejemplo, puede simplificarse en la forma:

```
rectype arb = record nombre: alfa;
    padre, madre: arb
end
```

¿Cuál es la imagen en memoria de la estructura recursiva correspondiente a la Figura 4.2?

Es de suponer que una implantación de este mecanismo de manejo de tipos recursivos utilizará un esquema de asignación dinámica de memoria, y los campos *padre* y *madre* del ejemplo tendrán dentro punteros generados automáticamente, e inaccesibles al programador. ¿Qué dificultades se presentarán al realizar esta implantación?

- 4.2. Definir la estructura de datos descrita en el último párrafo del Apartado 4.2 utilizando registros y punteros. ¿Se puede también representar este tipo de familia utilizando los tipos recursivos propuestos en el ejercicio anterior?
- 4.3. Supóngase que, utilizando una lista enlazada, se implementa una cola *Q* de elementos tipo *T*₀ (estructura «FIFO», es decir, «primero en entrar-primer en salir»). Definir una estructura de datos apropiada, procedimientos para insertar y extraer un elemento, y una función que compruebe si la cola está vacía o no. Los procedimientos deben tener su propio mecanismo de «reutilización» económica de la memoria.
- 4.4. Supóngase que los registros de una lista enlazada tienen una clave de tipo *entero*. Se pide escribir un programa que organice la lista en orden creciente de las claves, y construir un procedimiento que invierta el orden de la lista.
- 4.5. En las listas circulares se utiliza normalmente una *cabecera de lista* (ver Fig. 4.54). ¿Por qué se utiliza esta cabecera? Escribir procedimientos que hagan la inserción, el borrado y la búsqueda de un elemento identificado por una clave dada. Hacerlos primero asumiendo que hay una cabecera y, después, suponiendo que ésta no existe.

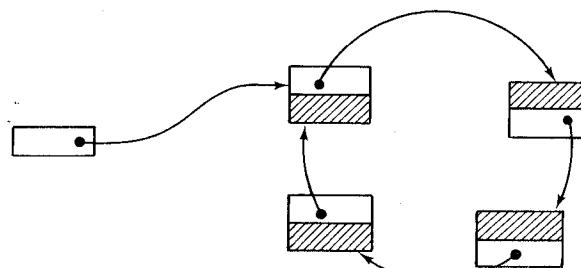


Fig. 4.54. Lista circular.

- 4.6. Una lista bidireccional es una lista de elementos enlazados en ambas direcciones (ver Fig. 4.55). Ambas enderezas de enlace parten de una *cabecera*. De forma análoga

al ejercicio anterior, se pide escribir un conjunto de procedimientos para buscar, insertar y borrar elementos en una lista de este tipo.

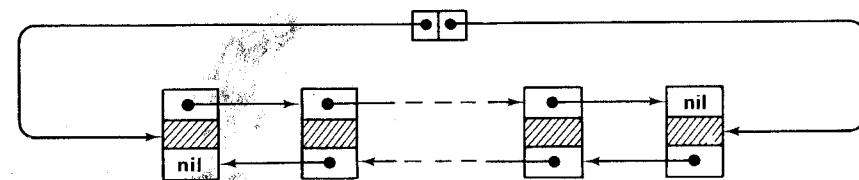


Fig. 4.55. Lista bidireccional.

- 4.7. ¿Funciona correctamente el Programa 4.2 si hay un par $\langle x, y \rangle$ en los datos que está repetido?
- 4.8. En muchos casos, el mensaje «ESTE CONJUNTO NO ESTA PARCIALMENTE ORDENADO» del Programa 4.2 no ayuda mucho. Ampliar el programa para que imprima una secuencia de los elementos que forman el ciclo, en el caso de que lo haya.
- 4.9. Escribir un programa que lea un texto de programa, identifique todas las definiciones de procedimientos (subrutinas) y las llamadas correspondientes, e intente establecer un orden topológico entre las subrutinas. Sea $P \prec Q$ si Q llama a P .
- 4.10. Dibujar el árbol construido por el Programa 4.3 con los $n + 1$ números:

$$n, 1, 2, 3, \dots, n$$

- 4.11. ¿Cuáles son las secuencias de nodos visitados al recorrer el árbol de la Fig. 4.23 en preorden, orden central y postorden?
- 4.12. Encontrar la regla de formación de una secuencia de n números tal que con el Programa 4.4, se obtenga un árbol perfectamente equilibrado.
- 4.13. Se definen los dos órdenes siguientes de recorrido de un árbol binario:
 - (a) (1) Recorrer el subárbol derecho.
(2) Visitar la raíz.
(3) Recorrer el subárbol izquierdo.
 - (b) (1) Visitar la raíz.
(2) Recorrer el subárbol derecho.
(3) Recorrer el subárbol izquierdo.

¿Hay relaciones simples entre las secuencias de nodos visitados según estos dos recorridos y las secuencias generadas con los recorridos definidos en el texto?

- 4.14. Definir una estructura de datos para representar árboles n -arios. A continuación escribir un procedimiento que recorra el árbol n -ario y construya un árbol binario con los mismos elementos. Suponiendo que la clave ocupa k palabras de memoria y el puntero una palabra, ¿qué mejora en la utilización del espacio se obtiene al usar un árbol binario en vez de un árbol n -ario?

- 4.15. Suponiendo que un árbol está definido como la estructura recursiva de datos (ver ejercicio 4.1):

```
rectype arbol = record x: integer;
    izquierdo, derecho: arbol
end
```

escribir un procedimiento que encuentre un elemento con clave dada x , y realice una operación P con él.

- 4.16. El catálogo de ficheros de un sistema está organizado como árbol binario ordenado. Cada nodo designa un fichero y especifica su nombre y , entre otras cosas, la fecha, codificada como número entero, en que fue utilizado por última vez.

Escribir un programa que recorra el árbol y borre todos los ficheros que han sido utilizados por última vez antes de una fecha determinada.

- 4.17. En una estructura árbol se mide empíricamente el número de accesos a cada elemento asociando un contador a cada nodo. Cada cierto tiempo, se actualiza la organización del árbol recorriendo el mismo, y produciendo un nuevo árbol según el Programa 4.4, con la inserción de las claves en orden de número de accesos decreciente. Escribir un programa que haga esta reorganización. ¿Es la longitud de camino media de este árbol igual, peor o mucho peor que la del árbol óptimo?

- 4.18. El método para analizar el algoritmo de inserción en un árbol descrito en el Apartado 4.5 puede utilizarse también para calcular los números esperados de comparaciones, C , y movimientos (intercambios), M , que realiza el método rápido (de ordenación, ver Programa 2.10) al ordenar N elementos de un array, suponiendo que todas las $n!$ permutaciones de las n claves $\{1, 2, \dots, n\}$ son igualmente probables. Encontrar la analogía y calcular C_n y M_n .

- 4.19. De todos los árboles equilibrados de 12 nodos, encontrar el que tiene la máxima altura. ¿En qué secuencia han de insersarse los nodos en el procedimiento (4.63) para producir este árbol?

- 4.20. Encontrar una secuencia de n claves que, al insertar con el procedimiento (4.63) haga que se realicen al menos una vez cada uno de los distintos reequilibrados (II , ID , DD , DI). ¿Cuál es el valor mínimo de n para una secuencia de tal tipo?

- 4.21. Encontrar un árbol equilibrado con claves $1 \dots n$, y una permutación de esas claves que aplicada al procedimiento de borrado (4.64), haga que se realicen al menos una vez cada una de las rutinas de reequilibrado. ¿Cuál es la secuencia de longitud, n , mínima?

- 4.22. ¿Cuál es la longitud de camino media del árbol de Fibonacci T_n ?

- 4.23. Escribir un programa que produzca un árbol casi óptimo utilizando el centroide como raíz (4.78).

- 4.24. Supóngase que se insertan las claves $1, 2, 3, \dots$ en un árbol B de orden 2 (Programa 4.7). ¿Qué claves originan divisiones de página? ¿Qué claves hacen que la altura del árbol crezca?

Si las claves se borran en el mismo orden, ¿qué claves dan lugar a unión (y eliminación) de páginas y qué claves hacen que la altura del árbol disminuya? Contestar

a la pregunta para (a) cuando se usa equilibrado (como en el Programa 4.7) y (b) cuando no se equilibran las páginas (al subocuparse una página, se toma un elemento de una página vecina).

- 4.25. Escribir un programa para buscar, insertar y borrar claves en un árbol B binario. Usar la definición de tipo de nodo (4.84). El método de inserción se ilustra en la Fig. 4.51.
- 4.26. Encontrar una secuencia de claves que al ser insertadas en un árbol B binario simétrico vacío, haga que el procedimiento (4.87) realice al menos una vez cada una de las operaciones de reequilibrado (II , ID , DD , DI). ¿Cuál es la secuencia más corta de las de este tipo?
- 4.27. Escribir un procedimiento para insertar elementos en un árbol B binario simétrico, encontrar un árbol, y una pequeña secuencia de claves que, al ser borradas del árbol, hagan que se realice al menos una vez cada una de las operaciones de reequilibrado.
- 4.28. Comparar los rendimientos de los algoritmos de inserción y borrado de elementos en árboles binarios, árboles equilibrados AVL, y árboles B binarios simétricos, en el computador de que disponga el lector. Estudiar, en especial, el efecto del empaquetamiento de datos, es decir, de elegir una representación que economice memoria a base de utilizar sólo dos bits por nodo para la información de equilibrio.
- 4.29. Modificar el algoritmo de impresión del Programa 4.6 para que sirva para dibujar árboles B binarios simétricos con arcos horizontales y verticales.
- 4.30. Si la cantidad de información asociada con cada clave es relativamente grande (comparada con la clave en sí), no debería almacenarse esta información dentro de la tabla hash. Explicar por qué y proponer un esquema para representar tal conjunto de datos.
- 4.31. Estudiar una solución al problema del «agrupamiento», a base de construir árboles de desbordamiento en vez de listas de desbordamiento, es decir, a base de organizar los elementos que han colisionado como estructuras árbol. Cada elemento de la tabla dispersa (hash) puede entonces considerarse la raíz de un árbol (posiblemente vacío) (hashing con árboles).
- 4.32. Diseñar un método que realice inserciones y *borrados* en una tabla hash utilizando incrementos cuadráticos para resolver las colisiones. Comparar experimentalmente este método con la organización de árbol binario simple, insertando y borrando al azar secuencias aleatorias de claves.
- 4.33. La principal desventaja de la técnica de tabla hash es que debe fijarse el tamaño de la tabla en un momento en que no se conoce el número real de elementos. Supóngase que hay un mecanismo de asignación dinámica de memoria, mediante el cual se puede obtener espacio adicional en cualquier momento. Así pues, cuando la tabla H está llena (o casi llena), se produce una tabla mayor H' , y se transfieren a ella todas las claves de H , poniendo a continuación H a disposición del mecanismo de gestión de memoria. Esta operación se llama «*rehashing*». Escribir un programa que la realice con una tabla H de dimensión n .
- 4.34. A menudo, las claves no son números enteros sino secuencias de letras, o palabras. Estas palabras suelen ser de longitud muy variable y, por lo tanto, no pueden ser almacenadas de forma económica y adecuada en campos de longitud fija. Escribir un programa que trabaje con una tabla hash y *claves de longitud variable*.

REFERENCIAS

- 4-1. ADELSON-VELSKII, G. M. y LANDIS, E. M., *Doklady Akademia Nauk SSSR*, **146**, (1962), 263-66; Traducción inglesa en *Soviet Math.*, 3, 1259-63.
- 4-2. BAYER, R. y MCCREIGHT, E., «Organization and Maintenance of Large Ordered Indexes», *Acta Informatica*, 1, No. 3 (1972), 173-89.
- 4-3. ———, «Binary B-trees for Virtual Memory», *Proc. 1971 ACM SIGFIDET Workshop*, San Diego, Nov. 1971, pp. 219-35.
- 4-4. ———, «Symmetric Binary B-trees: Data Structure and Maintenance Algorithms», *Acta Informatica*, 1, 4 (1972), 290-306.
- 4-5. HU, T. C. y TUCKER, A. C., *SIAM J. Applied Math.*, **21**, No. 4 (1971) 514-32.
- 4-6. KNUTH, D. E., «Optimum Binary Search Trees», *Acta Informatica*, 1, No. 1 (1971), 14-25.
- 4-7. MAURER, W. D., «An Improved Hash Code for Scatter Storage», *Comm. ACM*, **11**, No. 1 (1968), 35-38.
- 4-8. MORRIS, R., «Scatter Storage Techniques», *Comm. ACM*, **11**, No. 1 (1968), 38-43.
- 4-9. PETERSON, W. W., «Addressing for Random-access Storage», *IBM J. Res. & Dev.*, **1**, (1957), 130-46.
- 4-10. SCHAY, G. y SPRUTH, W., «Analysis of a File Addressing Method», *Comm. ACM*, **5**, No. 8 (1962), 459-62.
- 4-11. WALKER, W. A. y GOTLIEB, C. C., «A Top-down Algorithm for Constructing Nearly Optimal Lexicographic Trees», en *Graph Theory and Computing* (New York: Academic Press, 1972), pp. 303-23.

5

ESTRUCTURAS Y COMPILADORES DE LENGUAJES

El propósito de este capítulo es realizar un compilador (traductor) para un lenguaje de programación rudimentario y simple. Este programa compilador puede servir como ejemplo de desarrollo sistemático y estructurado, de un programa de tamaño y complejidad no triviales. Desde este punto de vista, constituye una buena aplicación de los mecanismos de estructuración de programas y datos vistos en los capítulos anteriores pero, además, se pretende dar una introducción general a la estructura y funcionamiento de los compiladores. El estudio y conocimiento de este tema sirve para comprender mejor el arte de la programación con lenguajes de alto nivel, y facilita al programador el desarrollo de sus propios sistemas, adecuados a distintos fines y áreas de aplicación. En lo referente al tema de los compiladores, el capítulo será forzosamente de carácter descriptivo y de introducción, habida cuenta de que la ingeniería de este tipo de programa es, evidentemente, un campo extenso y complicado. Quizás lo más importante es que la estructura de un lenguaje se refleja en la de su compilador, y su complejidad —o simplicidad— determina de manera precisa la complejidad de su compilador. Se empezará, por lo tanto, describiendo cómo se forman los lenguajes, y a continuación se prestará atención exclusivamente a estructuras lingüísticas sencillas que conduzcan a traductores modulares y simples. Estructuras lingüísticas de tal simplicidad son adecuadas prácticamente en todos los casos que se presentan en los lenguajes de programación reales.

5.1. DEFINICION Y ESTRUCTURA DE LOS LENGUAJES

Todo lenguaje está basado en un *vocabulario*. Sus elementos se llaman, normalmente, palabras; en el campo de los lenguajes formales, sin embargo, se llaman *íconos* (básicos). Es una característica de los lenguajes que algunas secuencias de palabras sean consideradas *frases* correctas del lenguaje, y otras sean

incorrectas o mal formadas. La gramática, sintaxis o estructura del lenguaje determina que una secuencia de palabras sea una frase correcta o no.

De hecho, se define la *sintaxis* como el conjunto de reglas o fórmulas que determinan el conjunto de frases (formalmente correctas). Sin embargo, tal conjunto de reglas no sólo permite decidir si una cierta secuencia de palabras es una frase, sino que, también, algo que es más importante, dotan a la frase de una estructura que ayuda a encontrar su significado. Resulta evidente por ello, que la sintaxis y la *semántica* (= significado) están íntimamente ligadas. Por lo tanto, las definiciones estructurales deben considerarse siempre como ayudas para un fin superior. Esto no debe impedir, sin embargo, estudiar inicialmente los aspectos estructurales de una forma exclusiva, sin considerar los aspectos de significado e interpretación.

Sea, por ejemplo, la frase «Pedro duerme». «Pedro» es el sujeto y «duerme» es el predicado. Esta frase pertenece al lenguaje que puede ser definido, por ejemplo, por la sintaxis siguiente:

$$\begin{aligned}\langle \text{frase} \rangle &::= \langle \text{sujeto} \rangle \langle \text{predicado} \rangle \\ \langle \text{sujeto} \rangle &::= \text{Pedro} \mid \text{Juan} \\ \langle \text{predicado} \rangle &::= \text{duerme} \mid \text{come}\end{aligned}$$

El significado de estas tres líneas es:

1. Una frase está formada por un sujeto seguido de un predicado.
2. Un sujeto es, bien la palabra «Pedro», o bien la palabra «Juan».
3. Un predicado es, bien la palabra «duerme», o bien la palabra «come».

La idea es que una frase puede deducirse a partir del *símbolo inicial* $\langle \text{frase} \rangle$, aplicando repetidamente las *reglas de sustitución*.

La notación con que se escriben estas reglas se llama BNF («*Backus-Naur Form*», es decir, forma de Backus y Naur). Fue utilizada por vez primera en la definición del ALGOL 60 [5-7]. Los objetos $\langle \text{frase} \rangle$, $\langle \text{sujeto} \rangle$ y $\langle \text{predicado} \rangle$ se llaman *símbolos no terminales*; *Pedro*, *Juan*, *duerme* y *come* se llaman *símbolos terminales*, y las reglas sintácticas *producciones*. Los símbolos $::=$ y \mid son *meta-símbolos* de la notación BNF. Si, para abreviar, se utilizan letras mayúsculas para los símbolos no terminales y letras minúsculas para los símbolos terminales, el ejemplo anterior puede escribirse en la forma:

EJEMPLO 1

$$\begin{aligned}I &::= AB \\ A &::= x \mid y \\ B &::= z \mid w\end{aligned}\tag{5.1}$$

y el lenguaje definido por esta sintaxis está formado por las cuatro frases *xz*, *yz*, *xw*, *yw*.

A efectos de mayor precisión, se introducen las siguientes definiciones matemáticas:

1. Sea un lenguaje $L = L(T, N, P, I)$ especificado por:
 - (a) Un vocabulario T de símbolos terminales.
 - (b) Un conjunto N de símbolos no terminales (categorías gramaticales).
 - (c) Un conjunto P de producciones (reglas sintácticas).
 - (d) Un símbolo I (incluido en N), llamado símbolo inicial.
2. El lenguaje $L(T, N, P, I)$ es el conjunto de secuencias de símbolos terminales, ξ , que pueden ser obtenidas a partir de I según la regla 3 descrita más adelante.

$$L = \{\xi \mid I \xrightarrow{*} \xi \text{ y } \xi \in T^*\} \tag{5.2}$$

(se utilizan letras del alfabeto griego para designar secuencias de símbolos). T^* designa el conjunto de secuencias de símbolos de T .

3. Se puede *generar* una secuencia σ_n a partir de una secuencia σ_0 si y sólo si existen secuencias $\sigma_1, \sigma_2, \dots, \sigma_{n-1}$ tales que cada σ_i puede ser generada directamente a partir de σ_{i-1} según la regla 4 descrita más adelante.

$$(\sigma_0 \xrightarrow{*} \sigma_n) \leftrightarrow ((\sigma_{i-1} \rightarrow \sigma_i) \text{ para } i = 1 \dots n) \tag{5.3}$$

4. Se puede *generar directamente* una secuencia η a partir de una secuencia ξ si y sólo si existen secuencias $\alpha, \beta, \zeta' y \eta'$ tales que:
 - (a) $\xi = \alpha \zeta' \beta$
 - (b) $\eta = \alpha \eta' \beta$
 - (c) P contiene la producción $\zeta' ::= \eta'$

Nota: Se utiliza $\alpha ::= \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$ como abreviatura del conjunto de producciones $\alpha ::= \beta_1, \alpha ::= \beta_2, \dots, \alpha ::= \beta_n$.

Por ejemplo, la secuencia *xz* del Ejemplo 1 puede obtenerse mediante la siguiente secuencia de pasos de generación directa: $I \rightarrow AB \rightarrow xB \rightarrow xz$; por tanto, $I \xrightarrow{*} xz$ y, como $xz \in T^*$, *xz* es una frase del lenguaje, es decir, *xz* $\in L$. Obsérvese que los símbolos no terminales *A* y *B* aparecen en pasos intermedios únicamente, mientras que el último paso debe conducir a una secuencia que contenga sólo símbolos terminales. Las reglas gramaticales se llaman producciones porque determinan cómo se pueden generar o producir nuevas formas o secuencias.

Se dice que un lenguaje es *libre de contexto* si y sólo si puede definirse por medio de un conjunto de producciones que sea libre de contexto. Un conjunto de producciones es libre de contexto si y sólo si todos sus miembros son de la forma

$$A ::= \xi \quad (A \in N, \quad \xi \in (N \cup T)^*)$$

es decir, el lado izquierdo es un solo símbolo no-terminal, A , y puede ser sustituido por ξ con independencia del contexto donde aparezca A . Si una producción tiene la forma

$$\alpha A \beta ::= \alpha \xi \beta,$$

se dice que es *sensible al contexto*, ya que la sustitución de A por ξ puede sólo realizarse en el contexto de α y β . En lo que sigue se considerarán únicamente sistemas libres de contexto.

El Ejemplo 2 muestra cómo, usando recurrencia, un conjunto finito de producciones puede generar un conjunto infinito de frases.

EJEMPLO 2

$$\begin{aligned} I &::= xA \\ A &::= z \mid yA \end{aligned} \quad (5.4)$$

A partir del símbolo I , pueden obtenerse las frases:

xz
 xyz
 $xyyz$
 $xyyyz$
 $\dots\dots$

5.2. ANALISIS DE FRASES

El objetivo primordial de los traductores de lenguajes no es la generación de frases sino el *reconocimiento* de ellas y de su estructura. Esto lleva consigo reconstruir, al leer una frase, los pasos que conducen a la generación de la misma, y seguirlos de nuevo. En el caso general, ésta es una tarea muy compleja y, algunas veces, incluso imposible. Su complejidad depende muy fuertemente del tipo de reglas sintácticas utilizadas para definir el lenguaje. La teoría del *análisis sintáctico* se encarga de desarrollar algoritmos de reconocimiento para lenguajes con reglas estructurales bastante complicadas. Aquí, sin embargo, sólo se pretende mostrar los puntos principales de un método que sirve para construir «reco-

nocedores» suficientemente sencillos y eficaces para ser utilizados en la práctica. Esto significa que el esfuerzo computacional para analizar una frase debe ser una función lineal de la longitud de la misma; en el caso más desfavorable (muy desfavorable), la función de dependencia puede ser $n \cdot \log n$, siendo n la longitud de la frase. Está claro que no hay que molestarse en buscar un algoritmo que reco-

noze cualquier lenguaje dado sino que, de manera pragmática, se trabajará en el sentido inverso: definir un algoritmo eficaz y, después, determinar el tipo de lenguajes que pueden ser tratados con él [5-3].

Una primera consecuencia de la eficacia requerida es que, en cada paso del análisis de una frase, la elección de qué hacer a continuación debe basarse sólo en el estado de la computación y en un único símbolo (el siguiente a leer). Otra consecuencia, de la mayor importancia, es que no se anule ningún paso realizado previamente. Normalmente, estas dos restricciones se conocen con el término técnico *un símbolo delante sin vuelta atrás*.

El método básico que se utilizará se llama análisis *descendente* («top-down»), porque intenta reconstruir los pasos de generación (que forman un árbol estructural en el caso más general) a partir del símbolo inicial, hasta llegar a la frase final, avanzando en la estructura desde arriba hacia abajo [5-5 y 5-6]. Para comenzar, considérese de nuevo el Ejemplo 1: Se supone dada la frase *Pedro come*, y hay que determinar si pertenece al lenguaje. Por definición, esto sucede sólo si la frase puede generarse partiendo del símbolo inicial *<frase>*. Según las reglas gramaticales, es evidente que sólo puede ser una frase si está compuesta de un sujeto seguido de un predicado. Partiendo de esto, la tarea pendiente de realizar puede dividirse en dos partes. Primero, se determina si alguna parte inicial de la frase puede generarse a partir del símbolo *<sujeto>*. Esto último resulta ser cierto ya que *Pedro* puede generarse directamente. Se borra el símbolo *Pedro* en la secuencia de entrada (es decir, se avanza la posición de lectura), y se continúa con la segunda parte: comprobar si lo que queda de la frase puede generarse a partir del símbolo *<predicado>*. Como esto resulta ser, a su vez, cierto, el resultado del proceso de análisis es afirmativo. Se puede ver el proceso descrito, en la figura que se muestra a continuación, donde se representa a la izquierda la tarea pendiente de realizar en cada momento, y a la derecha la parte de la secuencia de entrada que queda por leer.

<i><frase></i>	<i>Pedro come</i>
<i><sujeto><predicado></i>	<i>Pedro come</i>
<i>Pedro <predicado></i>	<i>Pedro come</i>
<i><predicado></i>	<i>come</i>
	<i>come</i>

Un segundo ejemplo ilustra el proceso de análisis de la frase *xyyz* utilizando las producciones del Ejemplo 2.

Como el proceso de seguir los pasos de generación de una frase se llama *partición*, lo que acaba de describirse es un *algoritmo de partición*. Desde luego, en los dos ejemplos vistos, cada uno de los pasos de sustitución puede decidirse en base a un único símbolo (el siguiente) de la secuencia de entrada. Desgraciadamente, esto no siempre es así, tal como se evidencia con el ejemplo 3:

I	$xyyz$
xA	$xyyz$
A	yyz
yA	yyz
A	yz
yA	yz
A	z
z	z
—	—

EJEMPLO 3

$$\begin{aligned} I &::= A|B \\ A &::= xA|y \\ B &::= xB|z \end{aligned} \quad (5.5)$$

Al intentar partir la frase $xxxxz$ se tiene

I	$xxxxz$
A	$xxxxz$
xA	$xxxxz$
A	xxz
xA	xxz
A	xz
xA	xz
A	z

y el proceso se atasca. La dificultad se presenta justo al dar el primer paso, ya que la decisión de sustituir I por A o B no puede tomarse en base únicamente al primer símbolo de la frase. Una solución puede ser elegir una de las dos opciones y, cuando el proceso se atasque, volver atrás hasta el punto donde se tomó la opción. Este proceso se llama *vuelta atrás*. Con el lenguaje del Ejemplo 3, no hay límite en el número de pasos que habría que «deshacer». Está claro que casos como éste deben evitarse; por lo tanto, en las aplicaciones prácticas se deben identificar, y evitar, lenguajes que requieran dar marcha atrás en el proceso de reconocimiento. Así pues, se considerarán únicamente sistemas gramaticales, que satisfagan la restricción de que los símbolos iniciales de partes derechas alternativas de cada producción sean distintos.

REGLA 1

Dada la producción

$$\Lambda ::= \xi_1 | \xi_2 | \dots | \xi_n$$

los conjuntos de símbolos iniciales de todas las frases que pueden ser generadas a partir de las ξ_i deben ser disjuntos, es decir:

$$\text{primero}(\xi_i) \wedge \text{primero}(\xi_j) = \emptyset \text{ para todo } i \neq j.$$

El conjunto $\text{primero}(\xi)$ está formado por todos los símbolos terminales que pueden aparecer en la primera posición de frases deducidas de ξ . Este conjunto se puede calcular según las dos reglas siguientes:

1. El primer símbolo del argumento es un símbolo terminal:

$$\text{primero}(a\xi) = \{a\}$$

2. El primer símbolo es un símbolo no terminal que tiene la siguiente regla de deducción:

$$A ::= \alpha_1 | \alpha_2 | \dots | \alpha_n$$

Entonces

$$\text{primero}(A\xi) = \text{primero}(\alpha_1) \cup \text{primero}(\alpha_2) \cup \dots \cup \text{primero}(\alpha_n)$$

En el Ejemplo 3 se observa que $x \in \text{primero}(A)$ y $x \in \text{primero}(B)$ y, por lo tanto, la primera producción viola la regla. Desde luego, es trivial encontrar una sintaxis, para el Ejemplo 3, que cumpla la Regla 1. La solución estriba en retrasar la «factorización» hasta que se hayan considerado todas las x . Las producciones siguientes son *equivalentes* a las (5.5) en el sentido de que generan el mismo conjunto de frases:

$$\begin{aligned} I &:= C | xI \\ C &:= y | z \end{aligned} \quad (5.5a)$$

Desgraciadamente, la Regla 1 no es lo suficientemente fuerte como para impedir que se presenten otros problemas. Considérese el

EJEMPLO 4

$$\begin{aligned} I &:= Ax \\ A &:= x | \epsilon \end{aligned} \quad (5.6)$$

donde ϵ designa la secuencia de símbolos nula. Al intentar analizar la frase x , se puede llegar al siguiente «callejón sin salida»:

I	x
Ax	x
xx	x
x	—

El problema surge porque debería haberse seguido la producción $A ::= \epsilon$, en vez de la $A ::= x$. Esta situación se llama el *problema de la secuencia nula*, y sólo se presenta cuando hay símbolos no-terminales que pueden generar la secuencia vacía o nula. Para evitarlo, se postula la

REGLA 2

Para todo símbolo $A \in N$ que genere la secuencia vacía ($A \xrightarrow{*} \epsilon$), el conjunto de sus símbolos iniciales debe ser disjunto del conjunto de símbolos que puedan seguir a cualquier secuencia obtenida a partir de A , es decir:

$$\text{primero}(A) \wedge \text{siguiente}(A) = \emptyset$$

El conjunto $\text{siguiente}(A)$ se calcula considerando todas las producciones P_i de la forma

$$X_i ::= \xi_i A \eta_i$$

y tomando el conjunto $C_i = \text{primero}(\eta_i) \cup \text{siguiente}(A)$ es la unión de todos esos conjuntos C_i . Si al menos un η_i es capaz de generar la secuencia vacía, entonces también debe incluirse $\text{siguiente}(X_i)$ en $\text{siguiente}(A)$. En el Ejemplo 4, el símbolo A viola la Regla 2, ya que:

$$\text{primero}(A) = \text{siguiente}(A) = \{\epsilon\}$$

La forma normal de especificar una repetición de símbolos es utilizar una definición recursiva. Por ejemplo, la producción

$$A ::= B \mid AB$$

describe el conjunto de secuencias B, BB, BBB, \dots Sin embargo, la Regla 1 impide ahora el uso de esta construcción, ya que:

$$\text{primero}(B) \wedge \text{primero}(AB) = \text{primero}(B) \neq \emptyset$$

Si se modifica ligeramente la producción, en la forma

$$A ::= \epsilon \mid AB$$

se generan las secuencias $\epsilon, B, BB, BBB, \dots$, y se viola la Regla 2, ya que

$$\text{primero}(A) = \text{primero}(B)$$

y, por lo tanto,

$$\text{primero}(A) \wedge \text{siguiente}(A) \neq \emptyset$$

Obviamente, las reglas prohíben el uso de definiciones que sean recursivas por la izquierda. Para evitar este tipo de definiciones, un método simple, que se utiliza con frecuencia, es usar recursión por la derecha:

$$A ::= \epsilon \mid BA$$

o extender el simbolismo de la notación BNF para poder expresar la repetición de símbolos explícitamente. Se hará esto último, suponiendo que $\{B\}$ designa el conjunto de secuencias

$$\epsilon, B, BB, BBB, \dots$$

Desde luego, habrá que tener presente que esta construcción puede generar la secuencia vacía. (Los paréntesis $\{ \}$ son meta-símbolos de la notación BNF ampliada.)

Se podría pensar, después de los razonamientos anteriores y de ver la transformación de las producciones (5.5) en (5.5a), que el «truco» de la transformación de gramáticas es la panacea universal para todos los problemas de análisis sintáctico. Hay que recordar, sin embargo, que la estructura de una frase ayuda a definir el significado de la misma, y que el significado de una construcción grammatical se explica normalmente en base al significado de sus partes componentes. Considérese, como ejemplo, el lenguaje de las expresiones aritméticas formadas por operandos a, b, c , y el símbolo menos significando la resta.

$$\begin{aligned} I &::= A \mid I - A \\ A &::= a \mid b \mid c \end{aligned}$$

Según esta gramática, la frase $a - b - c$ tiene una estructura que puede expresarse con ayuda de paréntesis como $((a - b) - c)$. Sin embargo, si la gramática se transforma en otra sintácticamente equivalente, pero sin recursión por la izquierda:

$$\begin{aligned} I &::= A \mid A - I \\ A &::= a \mid b \mid c \end{aligned}$$

entonces la misma frase (expresión aritmética) tiene otra estructura, es decir, la expresada por $(a - (b - c))$. Teniendo en cuenta el significado convencional de la resta, está claro que las dos formas no son semánticamente equivalentes en absoluto.

De acuerdo con lo anterior, la lección a retener es que, al definir un lenguaje que tenga un cierto significado inherente, hay que tener siempre en cuenta la *estructura semántica* al diseñar la estructura sintáctica, ya que esta última debe reflejar la anterior.

5.3. CONSTRUCCION DE UN GRAFO SINTACTICO

En el párrafo anterior se ha introducido un algoritmo de reconocimiento que es aplicable a gramáticas que satisfagan las restricciones de las Reglas 1 y 2. Ahora se estudia el problema de cómo convertir este algoritmo en un programa concreto. Se pueden aplicar dos técnicas que son esencialmente diferentes. Una es diseñar un programa analizador descendente que valga para cualquier gramática posible (que satisfaga las Reglas 1 y 2). En este caso, se codifica una gramática específica como una cierta estructura de datos que el programa lee, y éste opera en base a ella. Este analizador genérico está controlado en cierta forma por la estructura de datos; se dice entonces que el programa está *dirigido por tabla*. La otra técnica es diseñar un programa analizador descendente que sea específico para el lenguaje dado, y construirlo sistemáticamente según un conjunto de reglas que transforman una sintaxis dada en una secuencia de instrucciones, es decir, en un programa. Ambas técnicas tienen sus ventajas y sus desventajas y, en lo que sigue, se introducirán ambas. Cuando se desarrolla un compilador para un lenguaje de programación dado, apenas se necesita el alto grado de flexibilidad y parametrización que proporciona el analizador genérico y, sin embargo, un analizador específico conduce a sistemas más eficaces y manejables y es, por ello, preferible. En ambos casos, interesa representar la sintaxis dada por un grafo llamado *grafo sintáctico* o *de reconocimiento*. Este grafo refleja el flujo de control durante el proceso de análisis de una frase.

Una característica del método descendente es que el *objetivo* del proceso de análisis se conoce al comienzo del mismo. El objetivo es reconocer una frase, es decir, una secuencia de símbolos generable a partir del símbolo inicial. La aplicación de una producción, es decir, la sustitución de un símbolo único por una secuencia de símbolos, corresponde a la división de un objetivo único en un número de objetivos parciales, a conseguir en un orden específico. Por ello, el método descendente se conoce con el nombre de análisis *por objetivos*. Al construir un analizador, es fácil aprovechar esta correspondencia obvia entre símbolos no terminales y objetivos: se construye un analizador parcial para cada símbolo no terminal. Cada analizador parcial tiene el objetivo de reconocer una subfrase generable a partir de su correspondiente símbolo no-terminal. Como se desea construir un grafo que represente el analizador completo, se hará corresponder

un subgrafo con cada símbolo no-terminal. Esto conduce a las siguientes reglas para construir un grafo sintáctico.

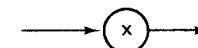
REGLAS DE CONSTRUCCION DEL GRAFO:

- A1. Para todo símbolo no terminal A que tenga un conjunto de producciones

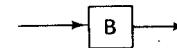
$$A ::= \xi_1 | \xi_2 | \dots | \xi_n$$

se construye un grafo sintáctico A cuya estructura viene determinada por la parte derecha de su conjunto de producciones según las reglas A2 a A6.

- A2. Cada aparición de un símbolo *terminal* x en un ξ_i corresponde a una instrucción que reconozca este símbolo seguida del avance de la posición de lectura al símbolo siguiente de la secuencia de entrada. Esto se representa en el grafo por un arco etiquetado con x dentro de un círculo.



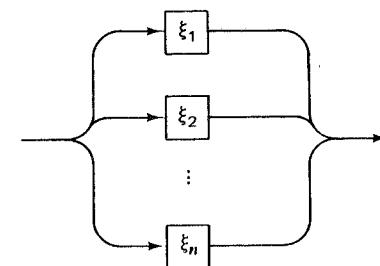
- A3. Cada aparición de un símbolo *no terminal* B en un ξ_i corresponde a una activación del reconocedor B . Esto se representa en el grafo con un arco etiquetado con B dentro de un cuadrado:



- A4. Para una producción de la forma

$$A ::= \xi_1 | \dots | \xi_n$$

se construye el grafo



donde cada ξ_i se obtiene aplicando las Reglas A2 – A6 a ξ_i .

A5. Para una ξ de la forma

$$\xi = \alpha_1 \alpha_2 \dots \alpha_m$$

se construye el grafo

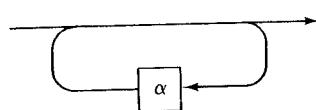


donde cada α_i se obtiene aplicando las Reglas A2 – A6 a α_i .

A6. Para un ξ de la forma

$$\xi = \{\alpha\}$$

se construye un grafo



donde α se obtiene aplicando las Reglas A2 – A6 a α .

EJEMPLO 5

$$\begin{aligned} A &::= x \mid (B) \\ B &::= AC \\ C &::= \{ + A \} \end{aligned} \tag{5.7}$$

Los símbolos terminales son $+$, x , $($ y $)$ mientras que $\{$ y $\}$ pertenecen al BNF ampliado y son, por lo tanto, meta-símbolos. El lenguaje que puede generarse a partir de A consta de expresiones con operando x , operador $+$ y paréntesis. Ejemplos de frases son:

x
 (x)
 $(x + x)$
 $((x))$
.....

Aplicando las seis reglas de construcción se obtienen los grafos de la Fig. 5.1. Obsérvese que pueden combinarse los grafos, para obtener uno sólo, mediante sustitución apropiada de C en B y de B en A (ver Fig. 5.2).

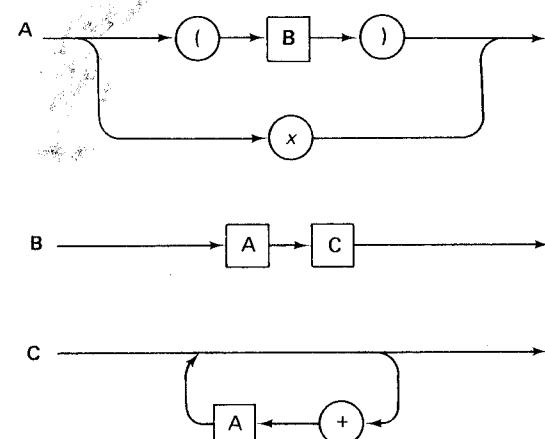


Fig. 5.1. Grafos sintácticos correspondientes al Ejemplo 5.

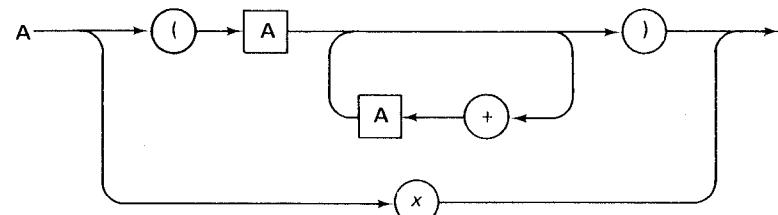


Fig. 5.2. Grafo sintáctico unificado correspondiente al Ejemplo 5.

El grafo sintáctico o de reconocimiento es una representación equivalente de la gramática del lenguaje; puede ser utilizado en lugar del conjunto de producciones en BNF. Es una representación de la sintaxis muy apropiada y, en muchos casos (si no todos), preferible a la BNF. Desde luego, el grafo presenta una imagen de la estructura del lenguaje más clara y concisa y, además, ayuda a entender mejor el proceso de análisis sintáctico. *El grafo es una forma adecuada para cuando se inicia el diseño de un lenguaje.* Se muestran ejemplos de especificaciones sintácticas de lenguajes completos en el Apartado 5.7, para PL/0, y en el Apéndice B, para PASCAL.

Las restricciones de las Reglas 1 y 2 fueron impuestas para poder realizar análisis sintáctico determinista con un único símbolo delante. ¿Cómo se manifiestan estas reglas en el grafo que representa la sintaxis? Es en este punto donde mejor se aprecia la claridad del grafo.

1. La Regla 1 se traduce en que, en cada nudo de ramificación, el camino a seguir debe poder seleccionarse en base únicamente a los primeros símbolos de las ramas. Esto implica que distintas ramas deben comenzar con símbolos distintos.
2. La Regla 2 se traduce en que, si hay un grafo A que puede ser recorrido sin leer ningún símbolo de entrada, hay que etiquetar esta «rama nula» con todos los símbolos que puedan seguir a A . (Esto afectará a la decisión a tomar al entrar en esta rama.)

Es fácil comprobar, sin acudir a la representación BNF de la gramática, si un sistema de grafos satisface o no estas dos reglas adaptadas. Como paso auxiliar se determinan los conjuntos $primero(A)$ y $siguiente(A)$ para cada uno de los grafos A . Aplicar las Reglas 1 y 2 es, entonces, inmediato. Un sistema de grafos que satisface estas dos reglas se llama un *grafo sintáctico determinista*.

5.4. CONSTRUCCION DE UN ANALIZADOR PARA UNA SINTAXIS DADA

A partir del grafo sintáctico determinista (si éste existe) de un lenguaje, es fácil deducir un programa que acepte y analice sintácticamente dicho lenguaje. El grafo representa, esencialmente, el flujo de control del programa. Sin embargo, al desarrollar éste, es muy importante seguir estrictamente un conjunto de reglas de traducción, similares a las utilizadas para obtener un grafo a partir de la representación BNF. Estas reglas se especifican a continuación, y son aplicables dentro de un marco específico. Este marco consta de un programa principal que contiene una rutina para avanzar al siguiente símbolo, y de los procedimientos correspondientes a los distintos objetivos parciales.

Para mayor simplicidad, supóngase que se representa la frase a tratar por el fichero *input* y que los símbolos terminales son caracteres individuales. Se postula la existencia de una variable *ch* que contiene siempre el siguiente símbolo a analizar. Se avanza al símbolo siguiente, entonces, con la instrucción:

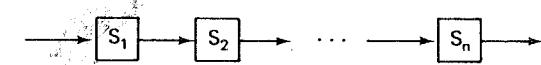
read(ch)

El programa principal está formado por una instrucción inicial que lee el primer carácter, seguida de una instrucción que activa el procedimiento del objetivo principal del análisis. Se obtienen las rutinas individuales correspondientes a los objetivos parciales o grafos sintácticos, aplicando las reglas siguientes. En lo que sigue, se denomina $T(S)$ la instrucción que se obtiene al traducir el grafo S .

REGLAS PARA TRADUCIR GRAFOS EN PROGRAMAS:

- B1. Por medio de sustituciones apropiadas, reducir el sistema de grafos a un número de grafos individuales lo más pequeño posible.

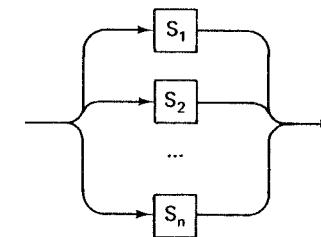
- B2. Traducir cada grafo resultante en una declaración de procedimiento, según las reglas siguientes B3 a B7.
- B3. Una secuencia de elementos



se traduce por la instrucción compuesta

```
begin T(S1); T(S2); ...; T(Sn) end
```

- B4. Una bifurcación de elementos



se traduce por la instrucción selectiva o condicional

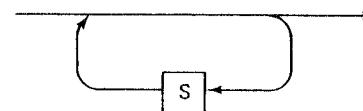
```
case ch of
  L1 : T(S1);
  L2 : T(S2);
  .....
  Ln : T(Sn)
end
```

```
if ch in L1 then T(S1) else
if ch in L2 then T(S2) else
.....
if ch in Ln then T(Sn) else
error
```

donde L_i designa el conjunto de símbolos iniciales de la construcción S_i ($L_i = primero(S_i)$).

Nota: si L_i está formado por un único símbolo a , entonces, desde luego, debe expresarse « $ch \in L_i$ » como « $ch = a$ ».

- B5. Un ciclo de la forma

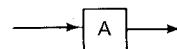


se traduce por la instrucción

while ch in L do T(S)

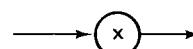
donde $T(S)$ es la traducción de S según las reglas B3 a B7, y L es el conjunto $L = \text{primero}(S)$ (ver nota anterior).

- B6. Un elemento del grafo que designe otro grafo A



se traduce por la instrucción de llamada a procedimiento, A .

- B7. Un elemento del grafo que designe un símbolo terminal x .



se traduce por la instrucción

if ch = x then read(ch) else error

donde *error* es una rutina que se activa cuando se encuentra un error sintáctico.

Se muestra la aplicación de estas reglas, traduciendo a continuación el grafo reducido del Ejemplo 5 (Fig. 5.2), por un programa de reconocimiento (Programa 5.1).

Programa 5.1. Analizador para la gramática del Ejemplo 5.

```
program analizar (input, output);
var ch: char;
procedure A;
begin if ch = 'x' then read(ch) else
      if ch = '(' then
      begin read(ch); A;
      while ch = '+' do
      begin read(ch); A
      end;
      if ch = ')' then read(ch) else error
      end else error
      end;
begin read(ch); A
end;
```

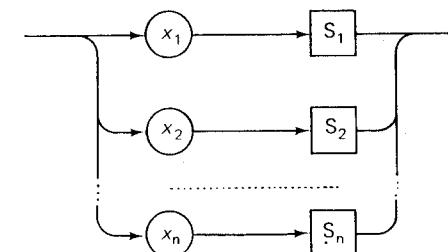
Al realizar la traducción se han aplicado algunas reglas de programación, que son obvias, para conseguir simplificar el programa. Una traducción literal hubiera dado como resultado, por ejemplo, para la cuarta línea:

**if ch = 'x' then
if ch = 'x' then read(ch) else error
else. . .**

que, obviamente, puede simplificarse tal como aparece en el programa. Las instrucciones de lectura en la quinta y séptima líneas provienen de simplificaciones similares.

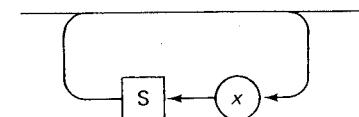
Parece razonable buscar dónde pueden hacerse tales simplificaciones, y representarlas directamente en forma de grafo. Las dos reglas adicionales siguientes cubren los dos casos de interés que se presentan:

B4a



**if ch = 'x₁' then begin read(ch); T(S₁) end else
if ch = 'x₂' then begin read(ch); T(S₂) end else
.....
if ch = 'x_n' then begin read(ch); T(S_n) end else error**

B5a



**while ch = 'x' do
begin read(ch); T(S) end**

Además, la forma

**read(ch); T(S);
while B do
begin read(ch); T(S) end**

que se repite con frecuencia, puede expresarse, desde luego, como

repeat *read(ch); T(S) until B* (5.8)

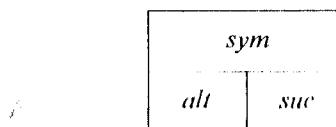
que es más corta.

El procedimiento *error* se ha dejado sin especificar a propósito. Como, de momento, el interés reside únicamente en saber si una frase de entrada está bien formada o no, se puede considerar este procedimiento como finalizador del programa. Naturalmente, en la práctica harán falta mejores métodos para tratar tales frases mal formadas. Este será el tema del Apartado 5.9.

5.5. CONSTRUCCION DE UN PROGRAMA ANALIZADOR DIRIGIDO POR TABLA

En vez de escribir un programa específico para cada lenguaje y sintaxis que se presente, según las reglas dadas en el apartado anterior, se puede construir un programa que sea un analizador genérico. En este caso, se suministran al programa genérico las gramáticas individuales, en forma de datos iniciales que preceden a las frases que vayan a analizarse. El programa sigue estrictamente las reglas de un simple método de análisis descendente, y resulta sencillo si el grafo sintáctico en cuestión es determinista, es decir, si la gramática es tal que permite analizar las frases con «un símbolo delante sin vuelta atrás».

Por lo tanto, la gramática, que se supone representada por un conjunto determinista de grafos sintácticos, se traduce en una estructura de datos adecuada, en vez de una estructura de programa [5-2]. La forma normal de representar un grafo es usar un nodo para cada símbolo y conectar los nodos por medio de punteros. Por ello, la tabla no es una simple estructura array. A continuación se dan las reglas para hacer esta traducción, que se justifican por sí mismas. Los nodos de la estructura de datos son registros de dos variantes: una para los símbolos terminales y otra para los no terminales. Los primeros se identifican por el símbolo terminal que representan, y los segundos por un puntero a la estructura de datos que representa el correspondiente símbolo no terminal. Ambas variantes contienen dos punteros, uno que designa el símbolo que sigue, el *sucesor*, y otro que forma la lista de posibles *alternativas*. La definición del tipo de datos resultante se presenta en (5.9). En los grafos, se dibujará un nodo como:



Resulta por otra parte, que se necesita también un elemento que represente la secuencia vacía, el símbolo nulo. Se designará por un elemento terminal llamado *vacio*.

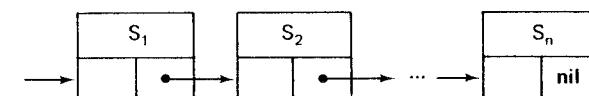
```
type puntero = ↑nodo;
nodo =
record suc, alt: puntero;
case terminal: boolean of
  true: (tsym: char);
  false: (nsym: punterocab)
end
```

(5.9)

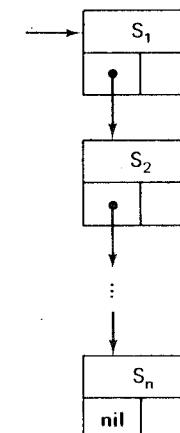
Las reglas de traducción de grafos en estructuras de datos son análogas a las Reglas B1 a B7.

REGLAS DE TRADUCCION DE GRAFOS EN ESTRUCTURAS DE DATOS:

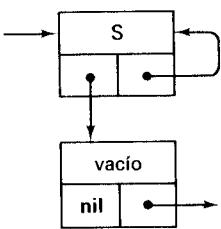
- C1. Por medio de sustituciones apropiadas, reducir el sistema de grafos a un número de grafos individuales tan pequeño como sea posible.
- C2. Traducir cada grafo resultante por una estructura de datos, según las reglas siguientes C3 a C5.
- C3. Una secuencia de elementos (ver figura de la Regla B3) se traduce por la siguiente lista de nodos de datos:



- C4. Una lista de alternativas (ver figura de la B4) se traduce por la estructura de datos



C5. Un ciclo (ver figura de la Regla B5) se traduce por la estructura



Como ejemplo, la estructura de la Fig. 5.3 corresponde al grafo de la sintaxis del Ejemplo 5 (Fig. 5.2). La estructura de datos se identifica con un nodo *cabecera* que contiene el nombre del símbolo no terminal (objetivo) correspondiente a la estructura. De momento la cabecera es innecesaria, ya que el puntero del campo objetivo bien podría apuntar directamente a la «entrada» de la estructura

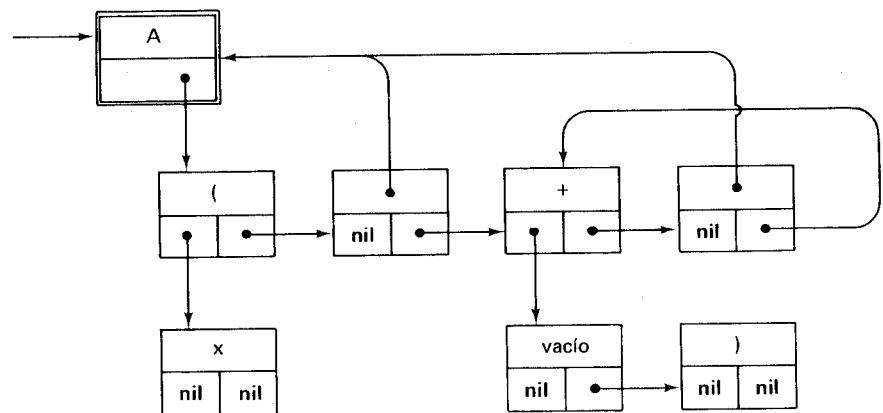


Fig. 5.3. Estructura de datos que representa al grafo de la Fig. 5.2.

correspondiente. Se puede utilizar la cabecera, sin embargo, para dar un nombre a la estructura.

```

type punterocab = ↑cabecera;
cabecera =
  record item: puntero;
  sym: char
  end
  
```

(5.10)

Ahora, el programa para analizar una frase —representada como una secuencia de caracteres en el fichero de entrada— está formado por una instrucción repetitiva que describe la transición de un nodo al siguiente. El programa

expresa como un procedimiento que describe la interpretación de un grafo; si se encuentra un nodo que representa un símbolo no terminal, entonces se interpreta ese grafo antes de acabar la interpretación del grafo en curso. Por lo tanto, el procedimiento de interpretación se activa *recursivamente*. Si el símbolo en curso (*sym*) en el fichero de entrada coincide con el símbolo del nodo en curso, entonces se escoge el campo *suc* para indicar el paso siguiente, si no se escoge el campo *alt*.

```

procedure analizar(objetivo: punterocab; var coincide: boolean);
var s: puntero;
begin s := objetivo↑ .item;
repeat
  if s↑ .terminal then
    begin if s↑ .tsym = sym then
      begin coincide := true; obtsym;
      end
    else coincide := (s↑ .tsym = vacio)
    end
  else analizar(s↑ .nsym, coincide);
  if coincide then s := s↑ .suc else s := s↑ .alt
until s = nil
end
  
```

(5.11)

El analizador (5.11) tiene la propiedad de que, cuando aparece un nuevo objetivo parcial *O*, trata de satisfacerlo inmediatamente sin comprobar previamente si el símbolo en curso está, o no, incluido en *primero(O)*. Esto implica que el grafo sintáctico en cuestión no debe tener elementos no terminales alternativos. En particular, si un símbolo no terminal es capaz de generar la secuencia vacía, ninguna de sus partes derechas debe comenzar con un símbolo no terminal.

A partir de (5.11), se pueden desarrollar analizadores dirigidos por tabla que sean más sofisticados y trabajen con clases de gramáticas más amplias. Sólo hacen falta pequeñas modificaciones, también, para realizar vuelta atrás, algo que, sin embargo, representa una pérdida de eficacia importante.

La representación de la sintaxis por medio de grafos tiene una desventaja cierta: los computadores no pueden leer grafos directamente. Sin embargo, antes de comenzar el análisis de las frases, hay que construir, de alguna manera, la estructura de datos que dirige el proceso. Es en este aspecto donde la representación BNF aparece como (forma) ideal, para suministrar la entrada de datos al programa analizador genérico. Por lo tanto, el siguiente apartado se dedica a la construcción de un programa que lee una secuencia de producciones BNF y las transforma, según las reglas B1 a B7, en una estructura interna de datos, con la cual el analizador (5.11) puede trabajar [5-8].

5.6. UN TRADUCTOR DE BNF EN ESTRUCTURAS DE DATOS PARA DIRIGIR ANALIZADORES

Un traductor que acepta producciones BNF y las convierte en otra representación distinta, es un verdadero ejemplo de programa cuyos datos de entrada pueden considerarse frases pertenecientes a un lenguaje. De hecho, resulta muy adecuado considerar la notación BNF como un lenguaje, caracterizado por su propia sintaxis, que puede a su vez, desde luego, especificarse por medio de producciones BNF. Como consecuencia, este traductor puede servir como un ejemplo adicional de construcción de un programa reconocedor que está, además, ampliado para ser un traductor o, expresado de forma más general, un procesador de sus datos de entrada. Por lo tanto, se procederá de la siguiente forma:

Paso 1. Definir una sintaxis del metalenguaje, llamado BNFA (BNF Ampliado).

Paso 2. Construir un reconocedor de BNFA según las reglas del Apartado 5.4.

Paso 3. Ampliar el reconocedor para obtener un traductor, por combinación con el analizador dirigido por tabla.

Supóngase que se define el meta-lenguaje —el lenguaje de las producciones sintácticas— por medio de las producciones siguientes:

$$\begin{array}{lcl} \langle \text{producción} \rangle & ::= & \langle \text{símbolo} \rangle = \langle \text{expresión} \rangle. \\ \langle \text{expresión} \rangle & ::= & \langle \text{término} \rangle \{, \langle \text{término} \rangle\} \\ \langle \text{término} \rangle & ::= & \langle \text{factor} \rangle \{ \langle \text{factor} \rangle\} \\ \langle \text{factor} \rangle & ::= & \langle \text{símbolo} \rangle \mid [\langle \text{término} \rangle] \end{array} \quad (5.12)$$

Obsérvese que se han utilizado meta-símbolos distintos de los BNF normales para el lenguaje de producciones a procesar. Hay dos razones para hacerlo:

1. Distinguir meta-símbolos y símbolos del lenguaje en (5.12).
2. Utilizar caracteres que se encuentran más fácilmente en los equipos de computación y, en particular, poder utilizar un único carácter, `:`, en vez de `::=`.

En la Tabla 5.1 se ilustran las correspondencias entre el BNF normal y la forma a utilizar para entrada de datos. Adicionalmente, todas las producciones no acaban con un punto.

BNF	BNFA de entrada
<code>::=</code>	<code>=</code>
<code> </code>	<code>,</code>
<code>{</code>	<code>[</code>
<code>}</code>	<code>]</code>

Tabla 5.1. Meta-símbolos y símbolos del lenguaje.

Utilizando este lenguaje BNFA para describir la sintaxis del Ejemplo 5 (5.7), se obtiene:

$$\begin{aligned} A &= x, (B). \\ B &= AC. \\ C &= [+A]. \end{aligned} \quad (5.13)$$

Para simplificar el traductor a construir, se postula que los símbolos terminales sean *letras aisladas* y que cada línea contenga únicamente una producción. Esto incluye la posibilidad de utilizar espacios en blanco en la entrada (para hacerla más legible) que son ignorados por el traductor. Sin embargo, la instrucción `read(ch)` de la Regla B7 debe sustituirse ahora por una llamada a una rutina que obtiene el siguiente carácter significativo. Esto es un caso muy simple de lo que suele llamarse un inspector léxico o, simplemente, un *inspector* («scanner»). El objetivo del inspector es extraer el siguiente *símbolo* —tal como está definido por las reglas que representan el lenguaje— a partir de la secuencia de *caracteres* de entrada. Hasta el momento se han considerado símbolos y caracteres como idénticos; esto es, sin embargo, un caso especial, y se hace rara vez en la práctica.

Como última regla, se postula que, en el BNF de entrada, los símbolos no terminales estén representados por las letras A a la H, y los símbolos terminales por las letras I a la Z. Esto es únicamente una regla que facilita el proceso y no tiene consecuencias importantes. Sirve para no tener que especificar los vocabularios de símbolos terminales y no terminales antes de especificar las producciones.

Procediendo estrictamente según las reglas de construcción de un analizador B1 a B7, después de comprobar que (5.12) cumple las restricciones de las Reglas 1 y 2, se obtiene el Programa 5.2 como reconocedor del lenguaje especificado por (5.12). Obsérvese que el inspector es `obtsym`.

Programa 5.2. Analizador para el lenguaje (5.12).

```
program analizador(input, output);
label 99;
const vacio = '*';
var sym: char;
procedure obtsym;
begin
repeat read(sym); write(sym) until sym ≠ '';
end {obtsym};
```

```

procedure error;
begin writeln;
writeln (' ENTRADA INCORRECTA '); goto 99
end {error};

procedure termino;
procedure factor;
begin
  if sym in ['A' .. 'Z', vacio] then obtsym else
    if sym = '[' then
      begin obtsym; termino;
        if sym = ']' then obtsym else error
      end else error
    end {factor};
begin factor;
  while sym in ['A' .. 'Z', '[', vacio] do factor
end {termino};

procedure expresion;
begin termino;
  while sym = ',' do
    begin obtsym; termino
    end
end {expresion};

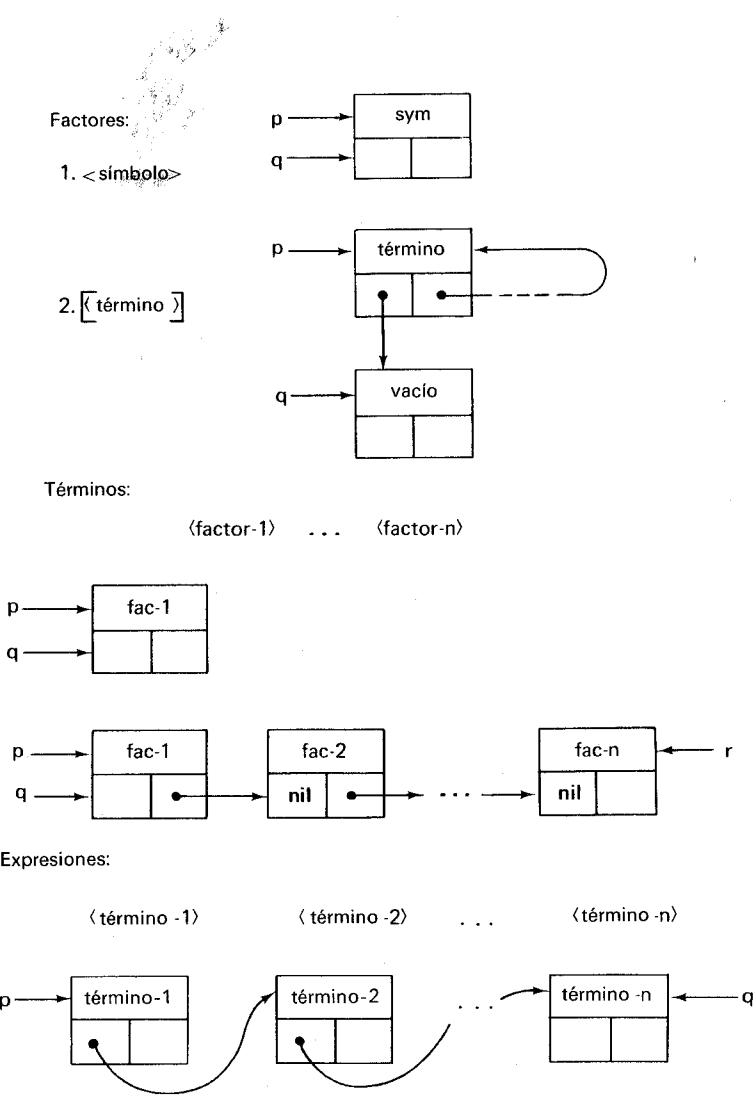
begin {programa principal}
  while not eof(input) do
    begin obtsym
      if sym in ['A' .. 'Z'] then obtsym else error;
      if sym = '=' then obtsym else error;
      expresion;
      if sym ≠ '.' then error;
      writeln; readln;
    end;
  99: end.

```

Programa 5.2. (Continuación)

El paso 3 del desarrollo del traductor es construir la estructura de datos, que se desea represente las producciones BNF leídas, que pueda ser interpretada por el procedimiento de análisis (5.11). Desgraciadamente, y al contrario de lo que sucedió con el paso de construcción de un reconocedor, este paso no puede formalizarse. A falta de una formalización, se describen nuevamente por medio de imágenes, las estructuras de datos que se quiere representen cada categoría gramatical del lenguaje. Estas estructuras se pasan como parámetros resultado a los procedimientos reconocedores correspondientes, que habrán sido ya amplia-

dos para convertirse en procedimientos traductores. Es natural devolver como resultados no las estructuras de datos en sí, sino los punteros *p*, *q* y *r*, que apuntan a ellas.



Está claro que es el procedimiento *factor* el que debe generar nuevos elementos de la estructura de datos; los otros dos procedimientos deberán enlazar los elementos según una lista lineal en la que *termino* utiliza el campo *suc* para hacer el

encadenamiento, y *expresion* el campo *alt*. Los detalles del proceso se hacen evidentes en el Programa 5.3.

La técnica que se utiliza para procesar símbolos no terminales necesita más aclaraciones. Un símbolo no terminal puede aparecer como factor de un término antes de presentarse a la izquierda de una producción. Se utiliza un procedimiento *encontrar(sym, c)* para localizar el símbolo *sym* en una lista lineal en la cual se agrupan todas las cabeceras que representan símbolos no terminales. Si se localiza el símbolo, se asigna a *c* una referencia al mismo; si no se encuentra en la lista, se añade. El procedimiento *encontrar* utiliza la técnica del centinela, vista en detalle en el Capítulo 4.

El Programa 5.3 tiene tres partes, y cada una de ellas corresponde a una sección de los datos de entrada. La parte 1 procesa las producciones, para obtener las estructuras de datos correspondientes. La parte 2 lee e identifica un único símbolo, aquél que se especifica como *símbolo inicial* para generar frases del lenguaje. (Va precedido de un signo \$ que separa la parte 1 de la parte 2 en los datos de entrada.) La parte 3 es el programa analizador (5.11) que lee *frases de entrada* bajo el control de la estructura de datos generada en la parte 1.

Es de destacar que el Programa 5.3 ha sido desarrollado únicamente insertando instrucciones adicionales en el Programa 5.2 que, por otra parte, no ha cambiado. Este último trata únicamente el reconocimiento de frases formadas correctamente, y puede utilizarse como un marco adecuado para el otro que, no sólo reconoce, sino que también procesa, o traduce, las frases que acepta. Este método de construcción de procesadores de lenguaje a través de *etapas de refinamiento* o, mejor, *enriquecimiento* por etapas, es muy recomendable. De esta forma, el diseñador puede estudiar un aspecto seleccionado del procesamiento del lenguaje antes de estudiar otros, y se facilita, por lo tanto, la verificación de la corrección del traductor o, por lo menos, el desarrollo del programa en un clima de gran confianza y seguridad. Este ejemplo es bastante simple y tiene sólo dos etapas. Lenguajes y traductores más complicados, requieren un número de pasos de refinamiento individuales bastante más alto. Un proceso de desarrollo similar, éste en tres pasos, será el objeto de los Apartados 5.8 a 5.11.

Tal como se evidencia durante el desarrollo del Programa 5.3, el método de *análisis sintáctico dirigido por tabla* —o mejor, dirigido por estructura de datos— proporciona un grado de flexibilidad y libertad que no tiene el analizador para un lenguaje específico. Aunque en general no se necesita, esta flexibilidad adicional es la verdadera esencia de los llamados *lenguajes ampliables*. Uno de estos lenguajes puede ser ampliado con formas sintácticas adicionales, más o menos a voluntad del programador. De forma análoga a los datos de entrada del Programa 5.3, los datos de entrada para un compilador de un lenguaje ampliable tienen una parte que especifica las ampliaciones al mismo a utilizar en el programa que viene a continuación. Un esquema incluso más ambicioso permite la modificación del lenguaje durante el proceso de traducción, mezclando partes del programa a traducir con secciones que especifican el nuevo lenguaje.

A pesar de lo interesante que estas ideas puedan parecer, los esfuerzos para

construir compiladores de este tipo han estado marcados por una considerable falta de éxito. La razón de ello está en que los aspectos sintácticos y de reconocimiento de frases son sólo una parte del proceso de traducción y, de hecho, una parte pequeña. Es también la parte que puede formalizarse más fácilmente y que, por lo tanto, puede ser representada más fácilmente por una estructura de tabla sistemática. El *significado* del lenguaje, es decir, el resultado de la traducción, es una parte mucho más difícil de formalizar. Este problema todavía no se ha resuelto siquiera a un nivel satisfactorio, lo cual explica por qué los diseñadores de compiladores, tienden a entusiasmarse con los lenguajes ampliables mucho más al principio de su primer proyecto, que al acabar el mismo. Como conclusión de esta lección, se dedica el resto de este capítulo a desarrollar un compilador modesto para un lenguaje de programación específico y simple.

Programa 5.3. Traductor del lenguaje (5.12).

```
program analizadorgenerico (input, output);
label 99;
const vacio = '*';
type puntero = ↑nodo;
punterocab = ↑cabecera;
nodo = record suc, alt: puntero;
case terminal: boolean of
  true: (tsym: char);
  false: (nsym: punterocab)
end;
cabecera = record sym: char;
  item: puntero;
  suc: punterocab
end;
var lista, centinela, cab: punterocab;
p: puntero;
sym: char;
ok: boolean;
procedure obtsym;
begin
repeat read(sym); write(sym) until sym ≠ '';
end {obtsym};
procedure encontrar(s: char; var cab: punterocab);
{localizar en la lista el simbolo no-terminal s; si no esta, insertarlo}
var cab1: punterocab;
begin cab1 := lista; centinela↑ .sym := s;
while cab1↑ .sym ≠ s do cab1 := cab1↑ .suc;
if cab1 = centinela then
  begin {insertar} new (centinela);
```

```

cab1↑ .suc := centinela; cab1↑ .item := nil
end;
cab := cab1
end {encontrar};

procedure error;
begin writeln;
writeln (' SINTAXIS INCORRECTA ');
end {error};

procedure term (var p, q, r: puntero);
var a, b, c: puntero;
procedure factor (var p, q: puntero);
var a, b: puntero; cab: punterocab;

begin if sym in ['A' .. 'Z', vacio] then
begin {simbolo} new(a);
if sym in ['A' .. 'H'] then
begin {noterminal} encontrar(sym, cab);
a↑ .terminal := false; a↑ .nsym := cab
end else
begin {terminal}
a↑ .terminal := true; a↑ .tsym := sym
end;
p := a; q := a; obtsym
end else
if sym = '[' then
begin obtsym; term(p, a, b); b↑ .suc := p;
new(b); b↑ .terminal := true; b↑ .tsym := vacio;
a↑ .alt := b; q := b;
if sym = ']' then obtsym else error
end else error
end {factor};

begin factor(p, a); q := a;
while sym in ['A' .. 'Z', '[', vacio] do
begin factor(a↑ .suc, b); b↑ .alt := nil; a := b
end;
r := a
end {term};

```

Programa 5.3. (Continuación)

```

procedure expresion (var p, q: puntero);
var a, b, c: puntero;
begin term(p, a, c); c↑ .suc := nil;
while sym = ',' do
begin obtsym;
term(a↑ .alt, b, c); c↑ .suc := nil; a := b
end;
q := a
end {expresion};

procedure analizar (objetivo: punterocab; var coincide: boolean);
var s: puntero;
begin s := objetivo↑ .item;
repeat
if s↑ .terminal then
begin if s↑ .tsym = sym then
begin coincide := true; obtsym
end
else coincide := (s↑ .tsym = vacio)
end
else analizar(s↑ .nsym, coincide);
if coincide then s := s↑ .suc else s := s↑ .alt
until s = nil
end {analizar};

begin {producciones}
obtsym; new(centinela); lista := centinela;
while sym ≠ '$' do
begin encontrar(sym, cab);
obtsym; if sym = '=' then obtsym else error;
expresion (h↑ .item, p); p↑ .alt := nil;
if sym ≠ '?' then error;
writeln; readln; obtsym
end;
cab := lista; ok := true; {ver si todos los simbolos estan definidos}
while cab ≠ centinela do
begin if cab↑ .item = nil then
begin writeln(' SIMBOLO INDEFINIDO ', cab↑ .sym);
ok := false
end;
cab := cab↑ .suc
end;

```

Programa 5.3. (Continuación)

```

if  $\neg$  ok then goto 99;
{simbolo objetivo}
obtsym; encontrar(sym, cab); readln; writeln;
{frases}
while  $\neg$  eof(input) do
begin write(''); obtsym; analizar(cab, ok);
  if ok  $\wedge$  (sym = '.') then writeln ('CORRECTA')
    else writeln ('INCORRECTA');
  readln
end;
99: end.

```

Programa 5.3. (Continuación)

5.7. EL LENGUAJE DE PROGRAMACION PL/0

El resto de este capítulo se dedica al desarrollo de un compilador para un lenguaje que va a llamarse PL/0. Este lenguaje debe ser tal que su compilador sea lo suficientemente pequeño como para poder presentarse en este libro y, al mismo tiempo, servir de ejemplo para estudiar los conceptos más importantes de la compilación de lenguajes de alto nivel. No hay duda de que podía haberse escogido un lenguaje más sencillo, o más complicado. PL/0 no es más que un posible compromiso entre algo suficientemente sencillo como para que la exposición resulte clara, y algo suficientemente complejo como para que el proyecto tenga interés. PASCAL, cuya sintaxis aparece en el Apéndice B, es un lenguaje considerablemente más complicado, y su compilador se ha desarrollado utilizando las mismas técnicas que van a exponerse a continuación.

En lo referente a estructuras de programa, PL/0 es bastante completo. Tiene, por supuesto, la instrucción de asignación, como forma constructiva básica al nivel de instrucciones. Los mecanismos de estructuración son los de concatenación, ejecución condicional y repetición (de instrucciones), representados por las formas conocidas, instrucciones **begin/end-**, **if-**, y **while-**. PL/0 posee también el concepto subrutina y, por ello, tiene declaración de procedimiento, y la correspondiente instrucción de llamada.

En cuanto a estructuras de datos, sin embargo, PL/0 se atiene sin compromiso a la necesidad de ser sencillo: los enteros son su único tipo de datos. Es posible declarar constantes y variables de este tipo. PL/0 tiene, por supuesto, los operadores relacionales y aritméticos convencionales.

La presencia de procedimientos, es decir, partes de un programa que están, más o menos, «autocontenidas», ofrece la oportunidad de introducir el concepto *ámbito* de un objeto (constante, variable y procedimiento). PL/0 tiene, por lo tanto, declaraciones al principio de los procedimientos, que indican que estos objetos han de entenderse como locales al procedimiento dentro del cual han sido declarados.

Con esta breve introducción, se puede entender de forma intuitiva la sintaxis

de PL/0. La Fig. 5.4 presenta esta sintaxis en forma de siete diagramas. El lector que esté interesado puede traducir estos diagramas a un conjunto de producciones BNF equivalente. La Fig. 5.4, que presenta la sintaxis de un lenguaje completo, de forma tan concisa y clara, es un buen ejemplo del poder expresivo de estos diagramas sintácticos.

A continuación se presenta un programa PL/0 que puede servir para ilustrar el uso de algunas de las partes de este mini-lenguaje. El programa contiene los algoritmos usuales para la multiplicación, división y encontrar el máximo común divisor (mcd) de dos números naturales.

Programa

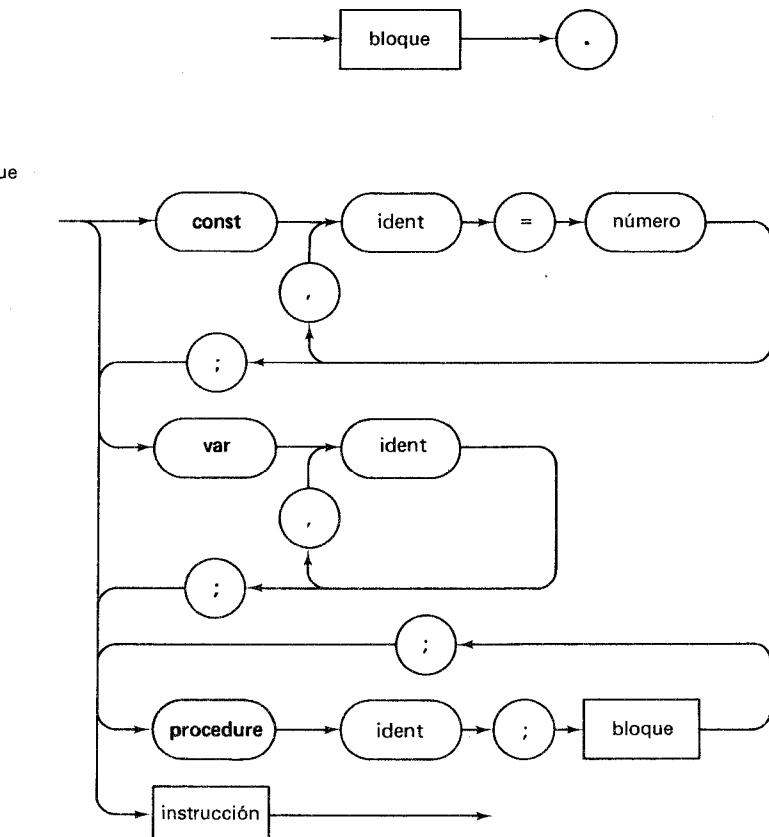


Fig. 5.4. Sintaxis del PL/0.

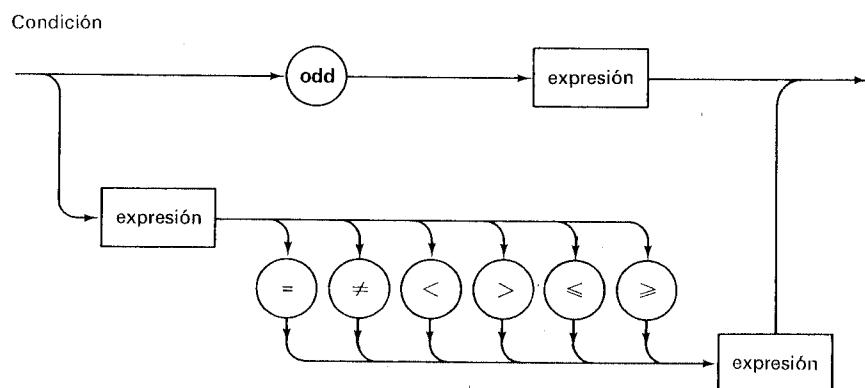
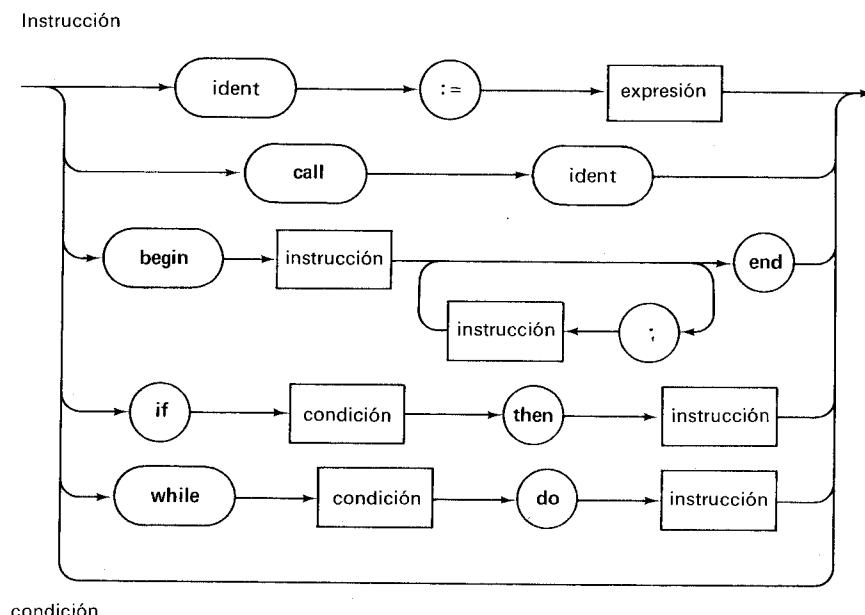


Fig. 5.4. (Continuación)

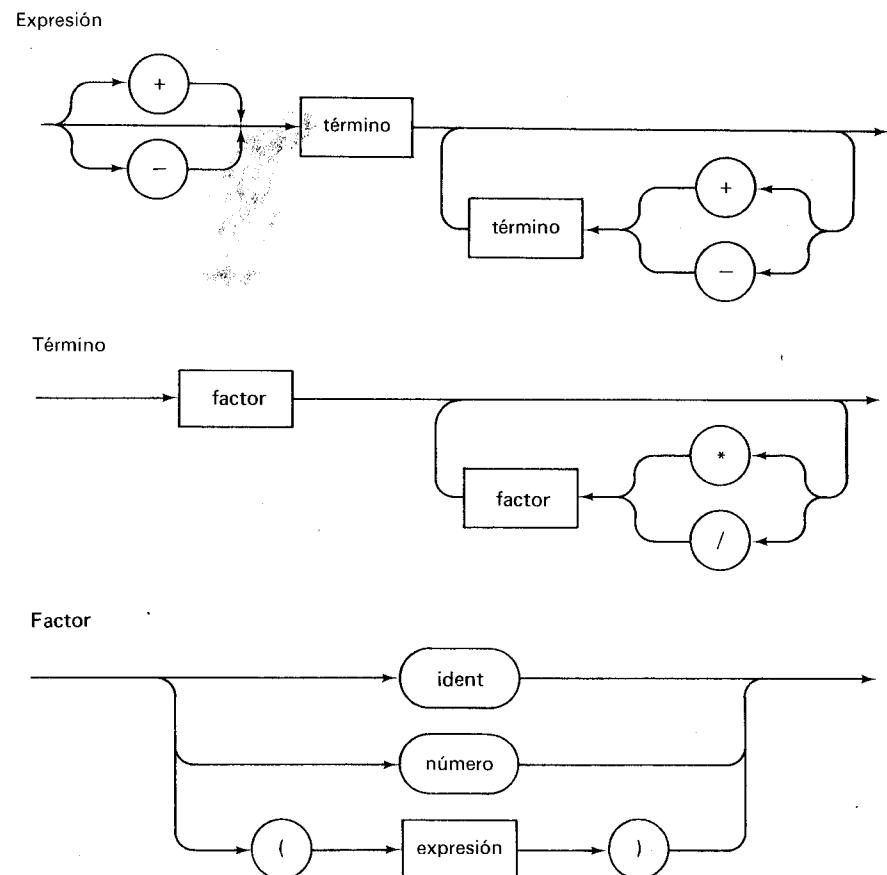


Fig. 5.4. (Continuación)

```

const m = 7, n = 85;
var x, y, z, q, r;
procedure multiplicar;
  var a, b;
  begin a := x; b := y; z := 0;
    while b > 0 do
      begin
        if odd b then z := z + a;
        a := 2 * a; b := b/2;
      end
    end;
  
```

(5.14)

```

procedure dividir;
  var w;
begin r := x; q := 0; w := y;
  while w ≤ r do w := 2 * w;
  while w > y do
    begin q := 2 * q; w := w/2;
    if w ≤ r then
      begin r := r - w; q := q + 1
      end
    end
end;

(5.15)

```

```

procedure mcd;
  var f, g;
begin f := x; g := y;
  while f ≠ g do
    begin if f < g then g := g - f;
           if g < f then f := f - g;
    end;
  z := f
end;

(5.16)

```

```

begin
  x := m; y := n; call multiplicar;
  x := 25; y := 3; call dividir;
  x := 84; y := 36; call mcd;
end.

```

5.8. UN ANALIZADOR SINTACTICO PARA PL/0

Como primer paso en el desarrollo de un compilador PL/0, se construye a continuación un analizador sintáctico. Esto puede hacerse siguiendo de forma estricta las Reglas B1 a B7 del Apartado 5.4. Este método, sin embargo, es sólo aplicable si la sintaxis cumple las Reglas de restricción 1 y 2. Hace falta, por lo tanto, comprobar que estas reglas, tal como se aplican a los grafos sintácticos, se cumplen en este caso.

La Regla 1 especifica que, siempre que en un grafo haya un punto de donde salen ramas alternativas, éstas deben tener símbolos iniciales distintos. Es muy fácil comprobar esto en los diagramas sintácticos de la Fig. 5.4. La Regla 2 se refiere a todos los grafos que pueden ser recorridos sin leer ningún símbolo. En la sintaxis de PL/0, el único grafo de este tipo es el que describe las instrucciones. La Regla 2 exige que todos los símbolos que puedan seguir a una instrucción, sean distintos de aquellos con los que pueda empezar una instrucción. Como más

adelante se van a necesitar los conjuntos de símbolos *primero* y *siguiente* de todos los grafos, se determinan ahora esos conjuntos para todos los siete símbolos no terminales (grafos), de la sintaxis del PL/0 (exceptuando «programa»).

La Tabla 5.2 da la respuesta deseada, es decir, que los conjuntos de símbolos iniciales y sucesores de instrucciones son disjuntos. Por lo tanto, es válido aplicar las Reglas B1 a B7 para construir el analizador sintáctico.

Símbolo no-terminal S_i	Símbolos iniciales $L(S_i)$	Símbolos sucesores $F(S_i)$
Bloque	const var procedure ident if call begin while ident call	. ; end
Instrucción	begin if while odd + -(ident numero	then do
Condición	+ -(ident numero	. ;) R
Expresión	ident numero (end then do
Término	ident numero (. ;) R + - *
Factor	ident numero (end then do

Tabla 5.2. Símbolos iniciales, y símbolos sucesores de PL/0.

El lector atento habrá observado que los símbolos básicos de PL/0 ya no son caracteres únicos, como ocurría en los ejemplos anteriores. En vez de ello, los símbolos básicos son secuencias de caracteres, tales como BEGIN o :=. Como ya se hizo en el Programa 5.3, se utiliza un inspector léxico para tratar los aspectos meramente representacionales o léxicos de la secuencia de símbolos de entrada. El inspector sirve para lo siguiente:

1. Saltar separadores (espacios en blanco).
2. Reconocer palabras reservadas, tales como BEGIN, END, etc.
3. Reconocer como identificadores las palabras que no son reservadas. El identificador en sí se asigna a una variable global llamada *id*.
4. Reconocer como números las secuencias de dígitos. El valor del número se asigna a una variable global llamada *num*.
5. Reconocer pares de caracteres especiales como, por ejemplo, :=.

Para inspeccionar la secuencia de caracteres de entrada, *obtsym* utiliza un procedimiento local, *obtch*, que obtiene el siguiente carácter. Aparte de esto, *obtch* también:

1. Reconoce y suprime los finales de línea.
2. Copia el fichero de entrada en el fichero de salida, produciendo así un listado del programa.
3. Imprime al principio de cada línea un número de orden o el valor del contador de posiciones.

El inspector proporciona el mecanismo «un símbolo delante» requerido. Además, el procedimiento auxiliar *obtch* representa un «carácter delante» adicional. Por lo tanto, este compilador trabaja, en total, con un símbolo, más un carácter, delante.

Los detalles de estas rutinas resultan evidentes en el Programa 5.4, que representa el analizador completo de PL/0. De hecho, este analizador ya está ampliado para almacenar en una *tabla* los identificadores declarados como constantes, variables y procedimientos. De esta forma, la presencia de un identificador dentro de una instrucción origina una búsqueda en esta tabla, que determina si el identificador ha sido declarado apropiadamente o no. La ausencia de tal declaración puede bien entenderse como un error sintáctico, ya que se ha cometido un error formal en la composición del texto del programa, al utilizar un símbolo «ilegal». El hecho de que, para detectar este error, haga falta utilizar una tabla donde se almacena información, es una consecuencia de la *dependencia del contexto* inherente al lenguaje, puesta de manifiesto en la regla que obliga a que todos los identificadores estén declarados en el contexto apropiado. De hecho, prácticamente todos los lenguajes de programación son dependientes del contexto en este sentido. Sin embargo, la sintaxis libre de contexto es uno de los modelos más valiosos para estos lenguajes, y ayuda mucho en la construcción sistemática de reconocedores para ellos. El esquema de trabajo así obtenido, puede ampliarse fácilmente para tratar los pocos elementos del lenguaje que son dependientes del contexto, tal como se evidencia con la introducción de la tabla de identificadores, en el analizador sintáctico que ahora se considera.

Antes de construir los procedimientos individuales de análisis, correspondientes a cada uno de los grafos sintácticos, es útil determinar las relaciones de dependencia entre estos grafos. A tal fin, se construye un *diagrama* llamado *de dependencia*, que muestra las relaciones entre los grafos individuales, es decir, lista, para cada grafo *G*, aquellos otros *G*₁, ..., *G*_n, en función de los que *G* está definido. También muestra qué procedimientos van a ser llamados por otros. La Fig. 5.5 muestra el grafo de dependencia de PL/0.

Los ciclos de la Fig. 5.5 indican dónde hay recursión. Por lo tanto, es muy importante que el lenguaje que se utilice para implementar el compilador PL/0, no esté limitado por la ausencia de recursión. Por otra parte, el diagrama de dependencia permite sacar conclusiones sobre la estructura jerárquica del programa analizador. Por ejemplo, todas las rutinas pueden estar dentro de (ser declaradas locales a) la rutina que analiza <programa> (que es, por lo tanto, el programa principal del analizador). Además, todas las rutinas debajo de <bloque>

pueden estar definidas como locales a la rutina que representa <bloque>. Naturalmente, todas estas rutinas utilizan *obtsym* que, a su vez, utiliza *obtch*.

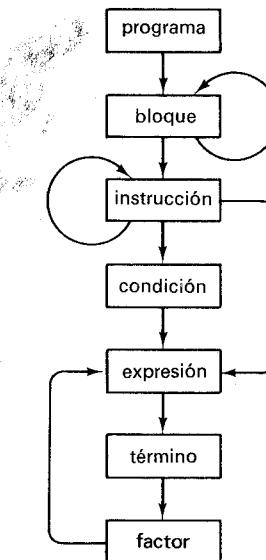


Fig. 5.5. Grafo de dependencia de PL/0.

Programa 5.4. Analizador sintáctico de PL/0.

```

program PL0 (input, output);
{compilador PL/0, análisis sintáctico únicamente}
label 99;
const nopr = 11; {no. de palabras reservadas}
      maxit = 100; {longitud de la tabla de identificadores}
      nmax = 14; {máximo no. de dígitos en los números}
      al = 10; {longitud de los identificadores}
type symbol =
      (nulo, ident, numero, mas, menos, por, barra, oddsym,
       igl, nig, mnr, mei, myr, mai, paren, parenc, coma, puntoycoma,
       punto, asignacion, beginsym, endsym, ifsym, thensym,
       whilesym, dosym, callsym, constsym, varsym, procsym);
alfa = packed array [1 .. al] of char;
objeto = (constante, variable, procedimiento);
var ch: char; {último carácter leído}
      sym: symbol; {último símbolo leído}
      id: alfa; {último identificador leído}
      num: integer; {último número leído}
      cc: integer; {contador de caracteres}
  
```

```

ll: integer;           {longitud de linea}
kk: integer;
linea: array [1 .. 81] of char;
a: alfa;
palabra: array [1 .. nopr] of alfa;
symp: array [1 .. nopr] of symbol;
syms: array [char] of symbol;
tabla: array [0 .. maxit] of
      record nombre: alfa;
          tipo: objeto
      end;

procedure error (n: integer);
begin writeln ('':cc,'^',n:2); goto 99
end {error};

procedure obtsym;
var i, j, k: integer;

procedure obtch;
begin if cc = ll then
  begin if eof(input) then
    begin write (' PROGRAMA INCOMPLETO '); goto 99
    end;
    ll := 0; cc := 0; write('');
  while not eoln(input) do
    begin ll := ll + 1; read(ch); write(ch); linea[ll] := ch
    end;
    writeln; ll := ll + 1; read(linea[ll])
  end;
  cc := cc + 1; ch := linea[cc]
end {obtch};
begin {obtsym}
while ch = ' ' do obtch;
if ch in ['A' .. 'Z'] then
begin {identificador o palabra reservada} k := 0;
repeat if k < al then
  begin k := k + 1; a[k] := ch
  end;
  obtch
until not (ch in ['A' .. 'Z', '0' .. '9']);
if k ≥ kk then kk := k else
  repeat a[kk] := ' '; kk := kk - 1
end;

```

Programa 5.4. (Continuación)

```

until kk = k;
id := a; i := 1; j := nopr;
repeat k := (i + j) div 2;
  if id ≤ palabra[k] then j := k - 1;
  if id ≥ palabra[k] then i := k + 1
until i > j;
if i - 1 > j then sym := symp[k] else sym := ident
end else
if ch in ['0' .. '9'] then
begin {numero} k := 0; num := 0; sym := numero;
repeat num := 10 * num + (ord(ch) - ord('0'));
  k := k + 1; obtch
until not (ch in ['0' .. '9']);
if k > nmax then error (30)
end else
if ch = ':' then
begin obtch;
  if ch = '=' then
    begin sym := asignacion; obtch
    end else sym := nulo;
end else
begin sym := syms[ch]; obtch
end
end {obtsym};

procedure bloque (it: integer);
procedure poner (k: objeto);
begin {poner el objeto en la tabla}
  it := it + 1;
  with tabla[it] do
    begin nombre := id; tipo := k;
    end
end {poner};

function posicion (id: alfa): integer;
var i: integer;
begin {encontrar en la tabla el identificador id}
  tabla[0].nombre := id; i := it;
  while tabla[i].nombre ≠ id do i := i - 1;
  posicion := i
end {posicion};

```

Programa 5.4. (Continuación)

```

procedure declaracionconst;
begin if sym = ident then
  begin obtsym;
  if sym = igl then
    begin obtsym;
    if sym = numero then
      begin poner (constante); obtsym
      end
    else error (2)
    end else error (3)
  end else error (4)
end {declaracionconst};

procedure declaracionvar;
begin if sym = ident then
  begin poner (variable); obtsym
  end else error (4)
end {declaracionvar};

procedure instrucion;
  var i: integer;
  procedure expresion;
    procedure termino;
      procedure factor;
        var i: integer;
        begin
          if sym = ident then
            begin i := posicion(id);
            if i = 0 then error (11) else
              if tabla[i] .tipo = procedimiento then error (21);
              obtsym
            end else
              if sym = numero then
                begin obtsym
                end else
                  if sym = paren then
                    begin obtsym; expresion;
                      if sym = parenc then obtsym else error (22)
                    end
                  else error (23)
                end {factor};
              begin {termino} factor;
                while sym in [por, barra] do

```

Programa 5.4. (Continuación)

```

begin obtsym; factor
end
end {termino};
begin {expresion}
if sym in [mas, menos] then
  begin obtsym; termino
  end else termino;
while sym in [mas, menos] do
  begin obtsym; termino
  end
end {expresion};
procedure condicion;
begin
  if sym = oddsym then
    begin obtsym; expresion
    end else
      begin expresion;
        if  $\neg$ (sym in [igl, nig, mnr, mei, myr, mai]) then
          error (20) else
            begin obtsym; expresion
            end
        end
      end {condicion};
begin {instrucion}
if sym = ident then
  begin i := posicion(id);
  if i = 0 then error (11) else
    if tabla[i] .tipo  $\neq$  variable then error (12);
    obtsym; if sym = asignacion then obtsym else error (13);
    expresion
  end else
    if sym = callsym then
      begin obtsym;
        if sym  $\neq$  ident then error (14) else
          begin i := posicion(id);
          if i = 0 then error (11) else
            if tabla[i] .tipo  $\neq$  procedimiento then error (15);
            obtsym
          end
        end else
          if sym = ifsym then
            begin obtsym; condicion;

```

Programa 5.4. (Continuación)

```

if sym = thensym then obtsym else error (16);
instrucion;
end else
if sym = beginsym then
begin obtsym; instrucion;
  while sym = puntoycoma do
    begin obtsym; instrucion
    end;
    if sym = endsym then obtsym else error (17)
  end else
if sym = whilesym then
begin obtsym; condicion;
  if sym = dosym then obtsym else error (18);
  instrucion
end
end {instrucion};

begin {bloque}
if sym = constsym then
begin obtsym; declaracionconst;
  while sym = coma do
    begin obtsym; declaracionconst
    end;
    if sym = puntoycoma then obtsym else error (5)
  end;
if sym = varsym then
begin obtsym; declaracionvar;
  while sym = coma do
    begin obtsym; declaracionvar
    end;
    if sym = puntoycoma then obtsym else error (5)
  end;
while sym = procsym do
begin obtsym;
  if sym = ident then
    begin poner (procedimiento); obtsym
    end
  else error (4);
  if sym = puntoycoma then obtsym else error (5);
  bloque (it);
  if sym = puntoycoma then obtsym else error (5);
end;

```

Programa 5.4. (Continuación)

```

instrucion
end {bloque};

begin {programa principal}
  for ch := 'A' to 'Z' do syms[ch] := nulo;
  palabra[1] := ' BEGIN ';
  palabra[3] := ' CONST ';
  palabra[5] := ' END ';
  palabra[7] := ' ODD ';
  palabra[9] := ' THEN ';
  palabra[11] := ' WHILE ';
  symp[1] := beginsym;
  symp[3] := constsym;
  symp[5] := endsym;
  symp[7] := oddsym;
  symp[9] := thensym;
  symp[11] := whylesym;
  syms['+'] := mas;
  syms['*'] := por;
  syms['('] := parenl;
  syms['='] := igl;
  syms ['.'] := punto;
  syms['<'] := mn;
  syms['≤'] := mei;
  syms[';'] := puntoycoma;
  page(output);
  cc := 0; ll := 0; ch := ' '; kk := al; obtsym;
  bloque (0);
  if sym ≠ punto then error (9);
99: writeln
end .

```

Programa 5.4. (Continuación)

5.9. TRATAMIENTO DE ERRORES SINTACTICOS

Hasta ahora, el analizador únicamente determina si una secuencia de símbolos de entrada pertenece al lenguaje o no. Como subproducto de este proceso, el analizador descubre también la estructura inherente en la frase. Pero, tan pronto como se detecta que una frase está mal construida, el analizador ya ha hecho su trabajo y el programa bien podría pararse. Desde luego, no es posible hacer esto en los compiladores reales. En vez de ello, un compilador debe dar un mensaje de error apropiado, y poder continuar el proceso de análisis —probablemente para encontrar otros errores—. Sólo se puede continuar, haciendo algún supuesto probable sobre la naturaleza del error y la intención del autor del programa in-

correcto, o saltando una parte de la secuencia de entrada, que venga a continuación, o haciendo ambas cosas. Hacer una suposición que tenga una probabilidad alta de ser correcta, es un arte muy difícil. Es algo que, hasta el momento, no ha podido ser formalizado, porque las formalizaciones de la sintaxis y del análisis sintáctico, no tienen en cuenta los muchos factores que influyen en forma importante en la mente humana. Por ejemplo, es un error común omitir símbolos de puntuación, tales como el punto y coma (¡y no sólo en programación!). En cambio, es bastante improbable que alguien olvide el signo + en una expresión aritmética. Para el analizador, los símbolos + y ; son sólo símbolos terminales, sin ninguna distinción adicional. Para el programador humano, sin embargo, el punto y coma apenas tiene significado y su uso al final de una línea parece redundante, mientras que el significado del operador aritmético aparece obvio sin lugar a dudas. Hay muchas otras consideraciones que deben tenerse en cuenta a la hora de diseñar un sistema adecuado de tratamiento de errores, y todas ellas dependen del lenguaje específico que se tenga, no pudiendo ser generalizadas a todos los lenguajes libres de contexto.

Sin embargo, se pueden postular algunas reglas e ideas que tienen validez más allá del campo de un sólo lenguaje como el PL/0. Se refieren tanto a la concepción inicial del lenguaje como al mecanismo de tratamiento de errores de su analizador léxico. Antes que nada, está bien claro que una *estructura de lenguaje sencilla* facilita, o incluso hace posible, un tratamiento de errores sensato. En particular, si se quiere saltar (ignorar) una parte del texto, cuando se encuentra un error, entonces es necesario que el lenguaje tenga *palabras clave*, cuyo uso inadecuado sea muy improbable, que puedan servir por lo tanto para que el analizador sintáctico se «estabilice». PL/0 sigue esta regla de una manera notable: todas las instrucciones estructuradas comienzan con una palabra clave definida, tales como **begin**, **if**, **while**, y lo mismo ocurre con los declaraciones, que empiezan con **var**, **const** o **procedure**. Esta regla se llamará, consecuentemente, la *regla de la palabra clave*.

La segunda regla se refiere, de forma más directa, a la construcción del analizador sintáctico. Es una característica de los analizadores descendentes que los objetivos se dividan en objetivos parciales y que unos analizadores llamen a otros para que analicen estos objetivos parciales. La segunda regla especifica que, cuando un analizador detecte un error, no debe únicamente pararse e informar del error al analizador que le ha llamado. En vez de ello, debe continuar la inspección del texto hasta un punto donde se pueda continuar algún análisis plausible. Esta regla se llamará, por lo tanto, la regla «antipánico». En cuanto a la programación del analizador, la consecuencia de esta regla es que no se podrá salir del mismo, a menos que sea por su punto de terminación normal.

Una posible interpretación estricta de esta regla es saltar texto, cuando se detecta una construcción ilegal, hasta el símbolo siguiente que pueda seguir correctamente a la estructura que se está analizando. Esto implica que cada analizador conozca el conjunto de símbolos que puedan seguir a su estructura sintáctica correspondiente, en el punto donde ha sido llamado.

Por lo tanto, en el primer paso de refinamiento (o enriquecimiento), se suministrará a cada procedimiento analizador un parámetro explícito, *symsig*, que especificará los posibles símbolos sucesores. Al final de cada procedimiento se pone una comprobación explícita de que el símbolo que viene a continuación en la secuencia de entrada está, de hecho, dentro del conjunto de los posibles sucesores (en el caso de que esta condición no esté ya fijada por la lógica del programa).

Sería, sin embargo, una decisión de diseño muy pobre, saltar el texto de entrada, en todas las circunstancias, hasta que se encontrara uno de tales símbolos sucesores. Después de todo, el programador puede haber omitido erróneamente un único símbolo (un punto y coma, por ejemplo), e ignorar todo el texto hasta donde esté el siguiente símbolo sucesor puede ser desastroso. Se amplían, por lo tanto, estos conjuntos de símbolos que terminan un posible salto de texto, con palabras clave que señalan, explícitamente, el comienzo de estructuras que no deben ser pasadas por alto. Por ello, los símbolos que se pasan como parámetros a los procedimientos analizadores son *símbolos de parada*, más que símbolos sucesores únicamente. Se pueden considerar los conjuntos de símbolos de parada como inicializados con diversos símbolos clave, siendo ampliados gradualmente con símbolos sucesores válidos, según se avanza en la jerarquía de objetivos parciales que se analizan. Para mayor flexibilidad, la comprobación descrita se realiza con una rutina genérica llamada *test*. Este procedimiento (5.17) tiene tres parámetros:

1. El conjunto de símbolos sucesores válidos, *c1*; si el símbolo en curso no está entre ellos, se tiene un error.
2. Un conjunto de símbolos de parada adicionales, *c2*, cuya presencia es, desde luego, un error, pero que no deben, en ningún caso, ser ignorados y saltados.
3. El número *n* del mensaje de error correspondiente.

```
procedure test (c1, c2: consym; n: integer);
begin if  $\neg$ (sym in c1) then
      begin error(n); c1 := c1 + c2;
           while  $\neg$ (sym in c1) do obtsym
           end
      end
end
```

(5.17)

El procedimiento (5.17) puede ser también convenientemente utilizado a la *entrada* de los procedimientos analizadores, para comprobar si el símbolo en curso es un símbolo inicial válido o no. Es recomendable hacer esto en todos los casos en que se llama incondicionalmente a un procedimiento analizador *X* como, por ejemplo, en la instrucción:

```
if sym = a1 then S1 else
    . . .
if sym = an then Sn else X
```

que es el resultado de traducir la producción sintáctica:

$$A ::= a_1 S_1 | \dots | a_n S_n | X \quad (5.18)$$

En estos casos, el parámetro $c1$ debe ser igual al conjunto de símbolos iniciales de X ; en cambio, $c2$ debe ser el conjunto de símbolos sucesores de A (ver Tabla 5.2). Los detalles de este procedimiento se encuentran en el Programa 5.5, que es la versión enriquecida del Programa 5.4. Para facilitar su lectura, se lista de nuevo el analizador completo, con la excepción del procedimiento *obtsym* y la inicialización de las variables globales, todo lo cual no sufre modificación alguna.

El esquema presentado hasta ahora tiene la propiedad de intentar la recuperación del analizador, al detectarse un error, ignorando uno o más símbolos del texto de entrada. Esta es una estrategia desacertada, en todos aquellos casos en que el error resulta de la *omisión* de un símbolo. La experiencia enseña que tales errores se restringen, casi completamente, a símbolos que tienen una función meramente sintáctica, sin representar una acción. Un ejemplo de ellos es el punto y coma de PL/0. Aumentar los conjuntos de símbolos sucesores con ciertas palabras clave hace, de hecho, que el analizador deje de saltar símbolos prematuramente. Esto equivale a comportarse como si se hubiera insertado uno de los símbolos que faltan. La parte de programa que analiza instrucciones compuestas, mostrada en (5.19), ilustra esto. El programa, de hecho, «inserta» los punto y coma que faltan delante de las palabras clave. El conjunto llamado *syminiinst* es el conjunto de símbolos iniciales de la estructura «instrucción».

```

if sym beginsym then
begin obtsym;
  instrucion([puntoycoma, endsym] + symsig);
  while sym in [puntoycoma] + syminiinst do
    begin
      if sym = puntoycoma then obtsym else error;
      instrucion([puntoycoma, endsym] + symsig)
    end;
    if sym = endsym then obtsym else error
  end
end
    
```

(5.19)

Se puede estimar el grado de éxito con que este programa diagnostica errores sintácticos, y se recupera en situaciones anormales, estudiando el programa PL/0 (5.20). El listado representa un resultado, producido por el Programa 5.5, y la Tabla 5.3 da un posible conjunto de mensajes de diagnóstico, correspondientes a los números de error del Programa 5.5.

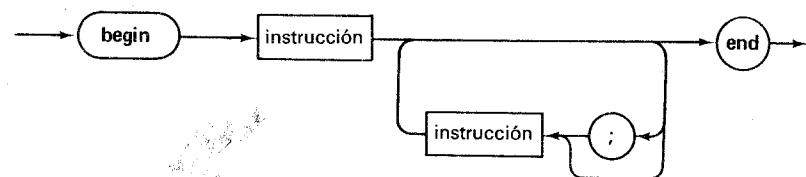


Fig. 5.6. Sintaxis modificada de la instrucción compuesta.

1. Usar = en vez de : =.
2. = debe ir seguido de un número.
3. El identificador debe ir seguido de =.
4. const, var y procedure deben ir seguidos de un identificador.
5. Falta una coma o un punto y coma.
6. Símbolo incorrecto después de una declaración de procedimiento.
7. Se esperaba una instrucción.
8. Símbolo incorrecto detrás de las instrucciones de un bloque.
9. Se esperaba un punto.
10. Falta un punto y coma entre instrucciones.
11. Identificador no declarado.
12. No están permitidas asignaciones a constantes o procedimientos.
13. Se esperaba el operador de asignación : =.
14. call debe ir seguido de un identificador.
15. No tiene sentido llamar a una constante o a una variable.
16. Se esperaba un then.
17. Se esperaba un end o un punto y coma.
18. Se esperaba un do.
19. Un símbolo incorrecto sigue a una instrucción.
20. Se esperaba un operador relacional.
21. Una expresión no debe contener un identificador de procedimiento.
22. Falta un paréntesis de cierre.
23. El factor anterior no puede ir seguido de este símbolo.
24. Una expresión no puede empezar con este símbolo.
30. Este número es demasiado grande.

Tabla 5.3. Mensajes de error del compilador PL/0.

El programa que viene a continuación ha sido obtenido introduciendo errores sintácticos en (5.14) a (5.16).

```

const m = 7, n = 85
var x, y, z, q, r;
  ↑ 5
  ↑ 5
procedure multiplicar;
  var a, b
  begin a := u; b := y; z := 0
  ↑ 5
  ↑ 11
  while b > 0 do
  ↑ 10
    
```

(5.20)

```

begin
  if odd b do z := z + a;
    ↑16
    ↑19
  a := 2a; b := b/2;
    ↑23
end
end;

procedure dividir
  var w;
    ↑5
  const two = 2, three = 3;
    ↑7
    ↑1
begin r = x; q := 0; w := y;
  ↑13
  ↑24
  while w ≤ r do w := two * w;
  while w > y
    begin q := (2 * q; w := w/2);
      ↑18
      ↑22
      ↑23
      if w ≤ r then
        begin r := r - w q := q + 1
          ↑23
        end
      end
    end;
end;

procedure mcd;
  var f, g;
begin f := x; g := y
  while f ≠ g do
    ↑17
    begin if f < g then g := g - f;
      if g < f then f := f - g;
      ↑21
      ↑22
      ↑23
      ↑24
    end;
end;

```

(5.20)

```

begin
  x := m; y := n; call multiplicar;
  x := 25; y := 3; call dividir;
  x := 84; y := 36; call mcd;
  call x; x := mcd; med = x
  ↑15
  ↑21
  ↑12
  ↑13
  ↑24
end.
  ↑17
  ↑5
  ↑7

```

PROGRAMA INCOMPLETO

Debería resultar claro después de lo visto, que no existe un esquema que traduzca sentencias correctas de forma razonablemente eficaz y, al mismo tiempo, sea capaz de tratar de forma adecuada todos los posibles errores sintácticos. Pero, ¡tampoco hace falta tanto! Cualquier esquema realizado con un esfuerzo razonable fallará, es decir, tratará de forma inadecuada algunos errores. Sin embargo, es importante que un buen compilador tenga las siguientes características:

1. Ninguna frase debe dar lugar a que el compilador pierda el control.
2. Todos los errores sintácticos que se presenten deben ser detectados y señalados.
3. Los errores que se presentan bastante frecuentemente y son verdaderos fallos del programador (debidos a descuidos o falta de comprensión), son diagnosticados correctamente y no hacen que el compilador tenga ningún (o muchos) tropiezo adicional (los llamados mensajes *no genuinos*).

El esquema presentado funciona satisfactoriamente, aunque las mejoras son siempre posibles. Su principal mérito es poder ser construido de forma sistemática a partir de unas pocas reglas básicas. Estas reglas se complementan meramente con algunas elecciones de parámetros, basadas en métodos heurísticos y en la experiencia de utilización del lenguaje.

Programa 5.5. Analizador de PL/0 con tratamiento de errores.

```

program PL0 (input, output);
{compilador PL/0 con tratamiento de errores sintácticos}
label 99;
const nopr = 11; {no. de palabras reservadas}
      maxit = 100; {longitud de la tabla de identificadores}
      nmax = 14; {máximo no. de dígitos en los números}

```

```

al = 10;           {longitud de los identificadores}
type symbol =
  (nulo, ident, numero, mas, menos, por, barra, oddsym,
   igl, nig, mnr, mei, myr, mai, paren, parenc, coma, puntoycoma,
   punto, asignacion, beginsym, endsym, ifsym, thensym,
   whilesym, dosym, callsym, constsym, varsym, procsym);
  alfa = packed array [1 .. al] of char;
  objeto = (constante, variable, procedimiento);
  consym = set of symbol;
var ch: char;          {ultimo caracter leido}
  sym: symbol;         {ultimo simbolo leido}
  id: alfa;            {ultimo identificador leido}
  num: integer;        {ultimo numero leido}
  cc: integer;          {contador de caracteres}
  ll: integer;          {longitud de linea}
  kk: integer;
  linea: array [1 .. 81] of char;
  a: alfa;
  palabra: array [1 .. nopr] of alfa;
  symp: array [1 .. nopr] of symbol;
  syms: array [char] of symbol;
  siminidecl, symininst, syminifac: consym;
  tabla: array [0 .. maxit] of
    record nombre: alfa;
      tipo: objeto
    end;
procedure error (n: integer);
begin writeln (' :cc, '^, n:2);
end {error};

procedure test (c1, c2: consym; n: integer);
begin if  $\neg$ (sym in c1) then
  begin error(n); c1 := c1 + c2;
    while  $\neg$ (sym in c1) do obtsym
  end
end {test};

procedure bloque (it: integer; symsig: consym);

procedure poner (k: objeto);
begin {poner el objeto en la tabla}
  it := it + 1;

```

Programa 5.5. (Continuación)

```

with tabla[it] do
  begin nombre := id; tipo := k;
  end
end {poner};

function posicion (id: alfa): integer;
  var i: integer;
begin {encontrar en la tabla el identificador id}
  tabla[0].nombre := id; i := it;
  while tabla[i].nombre  $\neq$  id do i := i - 1;
  posicion := i
end {posicion};

procedure declaracionconst;
begin if sym = ident then
  begin obtsym;
    if sym in [igl, asignacion] then
      begin if sym = asignacion then error (1);
        obtsym;
        if sym = numero then
          begin poner (constante); obtsym
          end
        else error (2)
        end else error (3)
      end else error (4)
    end {declaracionconst};

procedure declaracionvar;
begin if sym = ident then
  begin poner (variable); obtsym
  end else error (4)
end {declaracionvar};

procedure instrucion (symsig: consym);
var i: integer;

procedure expresion (symsig: consym);

procedure termino (symsig: consym);
procedure factor (symsig: consym);
  var i: integer;
begin test (syminifac, symsig, 24);
  while sym in syminifac do
  begin
    if sym = ident then
      begin i := posicion(id);

```

Programa 5.5. (Continuación)

```

if i = 0 then error (11) else
if tabla[i] .tipo = procedimiento then error (21);
    obtsym
end else
if sym = numero then
    begin obtsym
end else
if sym = parenta then
    begin obtsym; expresion ([parenc] + symsig);
        if sym = parenc then obtsym else error (22)
    end;
    test(symsig, [parenta], 23)
end
end {factor} ;
begin {termino} factor (symsig + [por, barra]);
    while sym in [por, barra] do
        begin obtsym; factor(symsig + [por, barra])
        end
    end {termino} ;
begin {expresion}
    if sym in [mas, menos] then
        begin obtsym; termino(symsig + [mas, menos])
        end else termino(symsig + [mas, menos]);
    while sym in [mas, menos] do
        begin obtsym; termino(symsig + [mas, menos])
        end
    end {expresion} ;
procedure condicion(symsig: consym);
begin
    if sym = oddsym then
        begin obtsym; expresion(symsig);
    end else
        begin expresion ([igl, nig, mnr, myr, mei, mai] + symsig);
            if  $\neg$ (sym in [igl, nig, mnr, mei, myr, mai]) then
                error (20) else
                    begin obtsym; expresion (symsig)
                    end
            end
    end {condicion} ;
begin {instrucion}
    if sym = ident then
        begin i : = posicion(id);

```

Programa 5.5. (Continuación)

```

if i = 0 then error (11) else
if tabla[i] .tipo  $\neq$  variable then error (12);
    obtsym; if sym = asignacion then obtsym else error (13);
    expresion(symsig)
end else
if sym = callsym then
    begin obtsym;
        if sym  $\neq$  ident then error (14) else
            begin i : = posicion(id);
                if i = 0 then error (11) else
                    if tabla[i] .tipo  $\neq$  procedimiento then error (15);
                    obtsym
                end
            end else
            if sym = ifsym then
                begin obtsym; condicion([thensym, dosym] + symsig)
                    if sym = thensym then obtsym else error (16);
                    instrucion(symsig);
            end else
            if sym = beginsym then
                begin obtsym; instrucion([puntoycoma, endsym] + symsig)
                    while sym in [puntoycoma] + syminiinst do
                        begin
                            if sym = puntoycoma then obtsym else error (10);
                            instrucion([puntoycoma, endsym] + symsig)
                        end;
                        if sym = endsym then obtsym else error (17)
                    end else
                    if sym = whilesym then
                        begin obtsym; condicion([dosym] + symsig);
                            if sym = dosym then obtsym else error (18);
                            instrucion(symsig);
                        end;
                        test(symsig, [], 19)
                    end {instrucion} ;
begin {bloque}
    repeat
        if sym = constsym then
            begin obtsym;
            repeat declaracionconst;
                while sym = coma do
                    begin obtsym; declaracionconst;
                end;

```

Programa 5.5. (Continuación)

```

if sym = puntoycoma then obtsym else error (5)
until sym ≠ ident
end;
if sym = varsym then
begin obtsym;
repeat declaracionvar;
while sym = coma do
begin obtsym; declaracionvar
end;
if sym = puntoycoma then obtsym else error (5)
until sym ≠ ident;
end;
while sym = procsym do
begin obtsym;
if sym = ident then
begin poner (procedimiento); obtsym
end
else error (4);
if sym = puntoycoma then obtsym else error (5);
bloque (it, [puntoycoma] + symsig);
if sym = puntoycoma then
begin obtsym; test(syminiinst + [ident, procsym], symsig, 6)
end
else error (5)
end;
test(syminiinst + [ident], syminidecl, 7)
until ¬(sym in syminidecl);
instruccion([puntoycoma, endsym] + symsig);
test(symsig, [ ], 8);
end {bloque};
begin {programa principal}

. . . Inicializacion (ver Programa 5.4) . . .

cc := 0; ll := 0; ch := ''; kk := al; obtsym;
bloque (0, [punto] + syminidecl + syminiinst);
if sym / punto then error (9);
99: writeln
end.

```

Programa 5.5. (Continuación)

6.10. UN PROCESADOR PL/0

Es realmente un hecho a destacar que, hasta aquí, el compilador PL/0 ha

sido desarrollado sin tener ningún conocimiento de la máquina para la que, se supone, va a generarse código. Pero, ¿por qué va a influir la estructura de la máquina objeto, en el esquema de análisis sintáctico y tratamiento de errores de un compilador? De hecho, *no debe* influir. En vez de ello, el esquema apropiado de generación de código para un computador cualquiera, se superpone al analizador existente, usando el método de refinamiento gradual.

Como se está a punto de hacer esto, es necesario elegir un procesador para el cual se va a compilar.

Para poder mantener la descripción del compilador razonablemente simple y libre de detalles marginales, debidos a peculiaridades de los procesadores reales, se postula la existencia de un computador, elegido para adaptarse a las características específicas de PL/0. Como este procesador no existe en la realidad (como sistema físico), es un procesador hipotético; se llamará *máquina PL/0*.

En este apartado no se pretende dar una explicación detallada de por qué se eligió precisamente este tipo de arquitectura de máquina. En vez de ello, el apartado se plantea como una descripción de la misma, formada por una introducción intuitiva seguida de una definición detallada del procesador, en forma de algoritmo. Esta formalización puede servir como ejemplo de cómo hacer descripciones detalladas y precisas de procesadores reales. El algoritmo interpreta instrucciones de la máquina PL/0 secuencialmente, y recibe el nombre de *intérprete*.

La máquina PL/0 está compuesta de dos memorias, un registro de instrucciones y tres registros de direcciones. La *memoria de programa*, llamada *código*, se carga por el compilador y permanece inalterada durante la interpretación del código. Puede considerarse, por lo tanto, como una memoria de sólo lectura. La *memoria de datos P* está organizada como una *pila*, y todos los operadores aritméticos trabajan con los dos elementos de la parte superior de la pila, sustituyendo los operandos por el resultado. El elemento superior se direcciona (indexa) con el *registro de la parte superior de la pila S*. El *registro de instrucciones I* contiene la instrucción que se está interpretando en cada momento. El *registro de dirección de programa D* designa la siguiente instrucción a interpretar.

En PL/0, los procedimientos pueden tener variables locales. Como los procedimientos pueden ser activados recursivamente, no puede asignarse memoria a estas variables antes de que se produzca la llamada a los mismos. Por ello, los segmentos de datos de cada uno de los procedimientos se apilan consecutivamente en la memoria tipo pila *P*. Como las activaciones de los procedimientos siguen de forma estricta el esquema «primero en entrar, último en salir», la pila es la estrategia de asignación de memoria apropiada. Cada procedimiento posee como propia cierta información interna, es decir, la dirección de programa desde donde fue llamado (llamada *dirección de retorno*), y la dirección del segmento de datos del módulo que le llamó. Estas dos direcciones son necesarias para proseguir adecuadamente la ejecución del programa, cuando se acabe el procedimiento. Pueden entenderse como variables internas, o locales implícitamente, situadas en el segmento de datos del procedimiento. Se llaman *dirección de retorno DR* y *enlace dinámico ED*. La dirección donde se encuentra el enlace dinámico, es

dicir, la dirección del segmento de datos creado más recientemente, se guarda en el *registro de dirección de base B*.

Como la asignación real de memoria tiene lugar en el momento de la ejecución (interpretación), el compilador no puede utilizar direcciones absolutas en el código que genera. Al no poder determinar más que la posición relativa de las variables dentro de un segmento de datos, sólo es capaz de generar *direcciones relativas*. El intérprete tiene que sumar a este (llamado) *desplazamiento* la dirección de base del segmento de datos apropiado. Si una variable es local al procedimiento que se está interpretando, la dirección de base correspondiente se encuentra en el registro B. Si no, ésta debe encontrarse descendiendo a lo largo de la cadena de segmentos de datos. El compilador, sin embargo, sólo puede conocer la profundidad estática de un camino de acceso, mientras la cadena de enlaces dinámicos mantiene la historia dinámica de las activaciones de procedimientos. Desgraciadamente, estos dos caminos de acceso no son necesariamente iguales.

Por ejemplo, supóngase que un procedimiento A llama a un procedimiento B, declarado local en A, B llama a C, declarado local en B, y C llama a B (recursivamente). Se dice que A está declarado al nivel 1, B al nivel 2 y C al nivel 3 (ver Fig. 5.7). Para acceder desde B, a una variable v declarada en A, el compilador sabe que existe una *diferencia de nivel* de 1 entre A y B. Sin embargo, al descender un paso en la cadena dinámica, se accedería a una variable local en C.

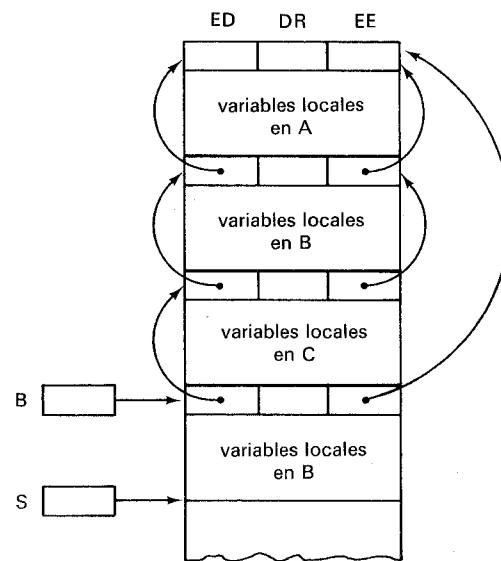


Fig. 5.7. Pila de la máquina PL/0.

Por ello, es evidente que se necesita una segunda cadena de enlaces, que rela-

cione los segmentos de datos en la forma que el compilador ve la situación. Se le denomina el *enlace estático EE*.

Por lo tanto, las direcciones se generan como pares de números, que indican la diferencia estática de nivel, y el desplazamiento dentro de un segmento de datos. Se supone que cada posición de la memoria de datos es capaz de almacenar una dirección o un entero.

El conjunto de instrucciones de la máquina PL/0 se ajusta para que sirva a las necesidades del lenguaje PL/0. Incluye las siguientes operaciones:

1. Una instrucción para cargar números (literales) sobre la pila (LIT).
2. Una instrucción para cargar variables sobre la pila (CAR).
3. Una instrucción de almacenamiento, correspondiente a las instrucciones de asignación (ALM).
4. Una instrucción para activar subrutinas, correspondiente a una llamada de procedimiento (LLA).
5. Una instrucción que asigna espacio en la memoria a base de incrementar el puntero de la pila, S(INS).
6. Instrucciones que realizan transferencia condicional e incondicional de control, usadas para las instrucciones «if» y «while» (SAL, SAC).
7. Un conjunto de operadores relacionales y aritméticos (OPR).

El formato de las instrucciones viene determinado por la necesidad de que tengan tres componentes, es decir, un código de operación f y un parámetro formado por una o dos partes (ver Fig. 5.8). En el caso de los operadores, el parámetro d determina el tipo de operador; en los otros casos es un número (LIT, INS), una dirección de programa (SAL, SAC, LLA), o una dirección de datos (CAR, ALM).

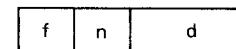


Fig. 5.8. Formato de las instrucciones.

Los detalles de operación de la máquina PL/0 deberían resultar evidentes a partir del procedimiento *interpretar*, que forma parte del Programa 5.6. Este combina el compilador completo junto con el intérprete, para formar un sistema que traduce programas PL/0 y, a continuación, los ejecuta. La modificación de este programa para que genere código para un computador existente, se deja como ejercicio al lector interesado. Puede considerarse la correspondiente ampliación del compilador, como una medida de lo apropiado que resulta, para la tarea que se está considerando, el computador elegido.

No cabe duda de que la máquina PL/0 vista, podría ampliarse con una organización más sofisticada, para que ciertas operaciones resultasen más eficientes. En particular, el mecanismo de direccionamiento elegido podía haber sido distinto. La solución presentada ha sido elegida debido a su sencillez, y porque todas las mejoras posibles deben estar basadas esencialmente en, y derivarse de, ella.

5.11. GENERACION DE CODIGO

Para poder formar una instrucción, el compilador debe conocer su código de operación y su parámetro, que es un número literal o una dirección. Es el mismo compilador quien asocia estos valores con los identificadores respectivos. La asociación se realiza al procesar las declaraciones de las constantes, variables y procedimientos. A tal fin, se amplía la tabla que contiene los identificadores, para almacenar estos atributos asociados con cada identificador. Si un identificador designa una constante, su atributo es el valor de la misma; si el identificador designa una variable, el identificador es su dirección, formada por un desplazamiento y un nivel; y si el identificador designa un procedimiento, sus atributos son la dirección donde comienza el procedimiento y su nivel. En el Programa 5.6 se muestra la correspondiente ampliación de la variable *tabla*. Constituye un ejemplo notable de un refinamiento (o enriquecimiento) de una declaración de datos, progresando de forma simultánea con el refinamiento gradual de las instrucciones.

Mientras el texto del programa suministra directamente los valores de las constantes, es tarea del compilador determinar las direcciones. PL/0 es lo suficientemente simple como para que la asignación secuencial de variables y código sea la solución obvia. Por lo tanto, cada declaración de una variable se procesará incrementando en una unidad un índice de asignación de memoria de datos (puesto que, por definición de la máquina PL/0, cada variable ocupa exactamente una posición de memoria). El índice de asignación de memoria de datos, *idat*, deberá inicializarse al comenzar la compilación de un procedimiento, reflejando el hecho de que su segmento de datos empieza vacío. [En realidad, *idat* toma inicialmente el valor 3, ya que todos los segmentos de datos contienen al menos las tres variables internas *DR*, *ED* y *EE* (ver apartado anterior)]. El procedimiento *poner*, que coloca los identificadores en la tabla, contiene los cálculos necesarios para determinar los atributos de cada identificador.

Teniendo esta información sobre los operandos, es bastante simple generar el código en sí. Debido a la adecuada organización, en forma de pila, de la máquina PL/0, existe prácticamente una correspondencia biunívoca entre operandos y operadores en el lenguaje fuente, e instrucciones en el código resultado. El compilador sólo tiene que hacer una reordenación apropiada, que produzca la notación *postfija*. Con «notación postfija» se quiere decir, que los operadores siempre vienen a continuación de sus operandos, en vez de ir entre ellos como en la notación convencional. La notación postfija se conoce a veces con el nombre de notación polaca (en honor de su inventor, Lukasciewicz), o notación *sin paréntesis*, debido a que hace que los paréntesis sean superfluos. En la Tabla 5.4 se muestran algunas correspondencias entre expresiones en notación postfija y notación convencional (ver también el Ap. 4.4.2).

Esta transformación se realiza con una técnica sencilla que se ilustra con los procedimientos *expresión* y *término* del Programa 5.6. Consiste simplemente en retrasar la transmisión del operador aritmético. Al llegar a este punto, el lector

Notación convencional	Notación postfija
$x + y$	$xy+$
$(x - y) + z$	$xy-z+$
$x - (y + z)$	$xyz+-$
$x*(y + z)*w$	$xyz+*w*$

Tabla 5.4. Expresiones en notación convencional y postfija.

debe comprobar que, con la organización que se tiene para los procedimientos analizadores, también se interpretan adecuadamente las reglas convencionales de prioridad de los distintos operadores.

La traducción de las instrucciones condicionales y repetitivas es, sin embargo, un poco más complicada. En este caso, se necesita a veces generar instrucciones de salto, para las que no se conoce todavía la dirección a saltar. Si se insiste en una generación de instrucciones estrictamente secuencial, en forma de un fichero de salida, se necesita un esquema de compilador en *dos pasos*. Es entonces el segundo paso el que se ocupa de suplementar las instrucciones de salto que están incompletas, con sus direcciones de destino. El compilador de este capítulo, sin embargo, utiliza una solución alternativa, consistente en colocar las instrucciones código en un array, y mantenerlas en memoria directamente accesible. Este método permite suministrar las direcciones que faltan tan pronto como se conocen. Esta operación se llama, normalmente, un «parche».

La única operación adicional a realizar, cuando se generan tales saltos hacia adelante, es guardar su emplazamiento, es decir, el índice que tienen en la memoria del programa. Este índice se utiliza, más adelante, para localizar la instrucción que hay que completar. Los detalles de este proceso también resultan evidentes en el Programa 5.6 (véanse las rutinas que procesan las instrucciones *if* y *while*). Los códigos generados para las instrucciones *if* y *while* tienen la forma siguiente (*E1* y *E2* significan direcciones en el código):

if <i>C</i> then <i>S</i>	while <i>C do S</i>
código para la condición <i>C</i>	<i>E1</i> : código para <i>C</i>
SAC <i>E1</i>	SAC <i>E2</i>
código para la instrucción <i>S</i>	código para <i>S</i>
<i>E1</i> : ...	SAL <i>E1</i>
	<i>E2</i> : ...

Por razones de comodidad, se introduce un procedimiento auxiliar llamado *gen*. Su función es ensamblar y emitir una instrucción formada con los valores de sus tres parámetros. También incrementa automáticamente el índice de código *ic*, que designa la posición de la siguiente instrucción a generar.

Como ejemplo, se lista a continuación, en forma nemotécnica, el código gene-

rado por el compilador al traducir la rutina de multiplicación (5.14). Los comentarios de la derecha han sido incluidos únicamente con fines explicativos.

```

2   INS  0,5    asignar espacio para enlaces y variables locales
3   CAR  1,3    x
4   ALM  0,3    a
5   CAR  1,4    y
6   ALM  0,4    b
7   LIT  0,0    0
8   ALM  1,5    z
9   CAR  0,4    b
10  LIT  0,0    0
11  OPR  0,12   >
12  SAC  0,29
13  CAR  0,4    b
14  OPR  0,7    odd
15  SAC  0,20
16  CAR  1,5    z
17  CAR  0,3    a
18  OPR  0,2    +
19  ALM  1,5    z
20  LIT  0,2    2
21  CAR  0,3    a
22  OPR  0,4    *
23  ALM  0,3    a
24  CAR  0,4    b
25  LIT  0,2    2
26  OPR  0,5    /
27  ALM  0,4    b
28  SAL  0,9
29  OPR  0,0    retornar

```

Código correspondiente al procedimiento PL/0 (5.14).

Muchas partes de la compilación de lenguajes de programación son considerablemente más complejas que las presentadas en el compilador PL/0 para la máquina PL/0 [5-4]. La mayor parte de ellas son difíciles de organizar de manera elegante. El lector que intente extender este compilador en una u otra dirección (hacia un lenguaje más potente, o hacia un computador más convencional), comprenderá rápidamente lo que hay de cierto en esta afirmación. Sin embargo, el método que se ha presentado aquí para desarrollar un programa complejo, sigue siendo válido e, incluso, tiene más interés, cuando la tarea a realizar se vuelve más sofisticada y compleja. De hecho, el método ha sido utilizado con éxito en la construcción de compiladores grandes ([5-1] y [5-9]).

Programa 5.6. Compilador PL/0.

```

program PL0(input, output);
{compilador PL/0 con generacion de codigo}
label 99;
const nopr = 11; {no. de palabras reservadas}
      maxit = 100; {longitud de la tabla de identificadores}
      nmax = 14; {maximo no. de digitos en los numeros}
      al = 10; {longitud de los identificadores}
      maxd = 2047; {maxima direccion}
      maxniv = 3; {maxima profundidad de imbricacion de los bloques}
      maxic = 200; {tamaño del array de codigo}
type symbol =
  (nulo, ident, numero, mas, menos, por, barra, oddsym,
  igl, nig, mnr, mei, myr, mai, paren, parenc, coma, puntoycoma,
  punto, asignacion, beginsym, endsym, ifsym, thensym,
  whilesym, dosym, callsym, constsym, varsym, procsym);
alfa = packed array [1 .. al] of char;
objeto = (constante, variable, procedimiento);
consym = set of symbol;
fcn = (lit, opr, car, alm, lla, ins, sal, sac); {funciones}
instrucion = packed record
  f: fcn; {codigo de funcion}
  ni: 0 .. maxniv; {nivel}
  di: 0 .. maxd; {desplazamiento}
end;
{ LIT 0,di : cargar la constante di
  OPR 0,di : ejecutar la operacion di
  CAR ni,di : cargar la variable ni, di
  ALM ni,di : almacenar la variable ni, di
  LLA ni,di : llamar el procedimiento di al nivel ni
  INS 0,di : incrementar el registro s en di
  SAL 0,di : saltar a di
  SAC 0,di : saltar condicionalmente a di }
var ch: char; {ultimo caracter leido}
sym: symbol; {ultimo simbolo leido}
id: alfa; {ultimo identificador leido}
num: integer; {ultimo numero leido}
cc: integer; {contador de caracteres}
ll: integer; {longitud de linea}
kk, err: integer;
ic: integer; {indice de codigo}
linea: array [1 .. 81] of char;
a: alfa;

```

```

codigo: array [0 .. maxic] of instrucion;
palabra: array [1 .. nopr] of alfa;
symp: array [1 .. nopr] of symbol;
syms: array [char] of symbol;
nemotecnico: array [fcn] of
    packed array [1 .. 5] of char;
syminidecl, syminiinst, syminifac: consym;
tabla: array [0 .. maxit] of
    record nombre: alfa;
        case tipo: objeto of
            constante: (val: integer);
            variable, procedimiento: (nivel, dir: integer)
        end;
procedure error(n: integer);
begin writeln(' ***', ': cc - 1, ↑, n: 2); err := err + 1
end {error};

procedure obtsym;
var i, j, k: integer;

procedure obtch;
begin if cc = ll then
    begin if eof(input) then
        begin write(' PROGRAMA INCOMPLETO '); goto 99
        end;
    ll := 0; cc := 0; write(ic: 5, '');
    while ¬eoln(input) do
        begin ll := ll + 1; read(ch); write(ch); linea[ll] := ch
        end;
    writeln; ll := ll + 1; read(linea[ll])
    end;
    cc := cc + 1; ch := linea[cc]
end {obtch};

begin {obtsym}
    while ch = ' ' do obtch;
    if ch in ['A' .. 'Z'] then
        begin {identificador o palabra reservada}      k := 0;
            repeat if k < al then
                begin k := k + 1; a[k] := ch
                end;
            obtch
        until ¬(ch in ['A' .. 'Z', '0' .. '9']);
    if k ≥ kk then kk := k else
        repeat a[kk] := ' '; kk := kk - 1
        until kk = k;
    id := a; i := 1; j := nopr;
    repeat k := (i + j) div 2;
        if id ≤ palabra[k] then j := k - 1;
        if id ≥ palabra[k] then i := k + 1
    until i > j;
    if i - 1 > j then sym := symp[k] else sym := ident
end else
    if ch in ['0' .. '9'] then
        begin {numero} k := 0; num := 0; sym := numero;
            repeat num := 10 * num + (ord(ch) - ord('0'));
            k := k + 1; obtch
        until ¬(ch in ['0' .. '9']);
        if k > nmax then error (30)
    end else
        if ch = ';' then
            begin obtch;
                if ch = '=' then
                    begin sym := asignacion; obtch
                    end else sym := nulo;
                end else
                    begin sym := syms[ch]; obtch
                    end
            end {obtsym};
procedure gen(x: fcn; y, z: integer);
begin if ic > maxic then
    begin write(' PROGRAMA DEMASIADO LARGO '); goto 99
    end;
with codigo[ic] do
    begin f := x; ni := y; di := z
    end;
    ic := ic + 1
end {gen};

procedure test (c1, c2: consym; n: integer);
begin if ¬(sym in c1) then
    begin error(n); c1 := c1 + c2;
        while ¬(sym in c1) do obtsym
    end
end {test};

```

Programa 5.6. (Continuación)

```

if k ≥ kk then kk := k else
    repeat a[kk] := ' '; kk := kk - 1
    until kk = k;
id := a; i := 1; j := nopr;
repeat k := (i + j) div 2;
    if id ≤ palabra[k] then j := k - 1;
    if id ≥ palabra[k] then i := k + 1
until i > j;
    if i - 1 > j then sym := symp[k] else sym := ident
end else
    if ch in ['0' .. '9'] then
        begin {numero} k := 0; num := 0; sym := numero;
            repeat num := 10 * num + (ord(ch) - ord('0'));
            k := k + 1; obtch
        until ¬(ch in ['0' .. '9']);
        if k > nmax then error (30)
    end else
        if ch = ';' then
            begin obtch;
                if ch = '=' then
                    begin sym := asignacion; obtch
                    end else sym := nulo;
                end else
                    begin sym := syms[ch]; obtch
                    end
            end {obtsym};
procedure gen(x: fcn; y, z: integer);
begin if ic > maxic then
    begin write(' PROGRAMA DEMASIADO LARGO '); goto 99
    end;
with codigo[ic] do
    begin f := x; ni := y; di := z
    end;
    ic := ic + 1
end {gen};

procedure test (c1, c2: consym; n: integer);
begin if ¬(sym in c1) then
    begin error(n); c1 := c1 + c2;
        while ¬(sym in c1) do obtsym
    end
end {test};

```

Programa 5.6. (Continuación)

```

procedure bloque(niv, it: integer; symsig: consym);
  var idat: integer; {indice de asignacion de memoria de datos}
    it0: integer; {indice inicial de la tabla}
    ic0: integer; {indice inicial de codigo}
  procedure poner(k: objeto);
  begin {poner objeto en la tabla}
    it := it + 1;
    with tabla[it] do
      begin nombre := id; tipo := k;
        case k of
          constante: begin if num > maxd then
            begin error (30); num := 0 end;
            val := num
          end;
          variable: begin nivel := niv; dir := idat; idat := idat + 1;
            end;
          procedimiento: nivel := niv
            end;
        end
      end {poner};
  function posicion(id: alfa): integer;
    var i: integer;
  begin {encontrar el identificador id en la tabla}
    tabla[0].nombre := id; i := it;
    while tabla[i].nombre ≠ id do i := i - 1;
    posicion := i
  end {posicion};
  procedure declaracionconst;
  begin if sym = ident then
    begin obtsym;
      if sym in [igl, asignacion] then
        begin if sym = asignacion then error(1);
          obtsym;
          if sym = numero then
            begin poner(constante); obtsym
              end
            else error (2)
          end else error (3)
        end else error (4)
      end {declaracionconst};
  procedure declaracionvar;

```

Programa 5.6. (Continuación)

```

begin if sym = ident then
  begin poner(variable); obtsym
  end else error (4)
end {declaracionvar};
procedure listarcodigo;
  var i: integer;
begin {listar el código generado para este bloque}
  for i := ic0 to ic - 1 do
    with codigo[i] do
      writeln(i, nemotecnico[f]: 5, ni : 3, di : 5)
end {listarcodigo};
procedure instruccion(symsig: consym);
  var i, ic1, ic2: integer;
  procedure expresion(symsig: consym);
    var opsuma: symbol;
    procedure termino(symsig: consym);
      var opmul: symbol;
    procedure factor(symsig: consym);
      var i: integer;
    begin test(symminifac, symsig, 24);
      while sym in symminifac do
        begin
          if sym = ident then
            begin i := posicion(id);
              if i = 0 then error (11) else
                with tabla[i] do
                  case tipo of
                    constante: gen(lit, 0, val);
                    variable: gen(car, niv-nivel, dir);
                    procedimiento: error (21)
                  end;
                  obtsym
                end else
                  if sym = numero then
                    begin if num > maxd then
                      begin error (30); num := 0
                      end;
                      gen(lit, 0, num); obtsym
                    end else
                      if sym = paren then
                        begin obtsym; expresion([parenc] + symsig);
                          if sym = parenc then obtsym else error (22)
                        end;

```

Programa 5.6. (Continuación)

```

    test(symsig, [parenc], 23)
  end
end {factor};
begin {termino} factor(symsig + [por, barra]);
  while sym in [por, barra] do
    begin opmul := sym; obtsym; factor(symsig + [por, barra]);
      if opmul = por then gen(opr, 0, 4) else gen(opr, 0, 5)
    end
  end {termino};
begin {expresion};
  if sym in [mas, menos] then
    begin opsuma := sym; obtsym; termino(symsig + [mas, menos]);
      if opsuma = menos then gen(opr, 0, 1)
      end else termino(symsig + [mas, menos]);
  while sym in [mas, menos] do
    begin opsuma := sym; obtsym; termino(symsig + [mas, menos]);
      if opsuma = mas then gen(opr, 0, 2) else gen(opr, 0, 3)
    end
  end {expresion};
procedure condicion(symsig: consym);
  var oprel: symbol;
begin
  if sym = oddsym then
    begin obtsym; expresion(symsig); gen(opr, 0, 6)
  end else
    begin expresion([igl, nig, mnr, myr, mei, mai] + symsig);
      if  $\neg$ (sym in [igl, nig, mnr, mei, myr, mai]) then
        error (20) else
      begin oprel := sym; obtsym; expresion(symsig);
        case oprel of
          igl: gen(opr, 0, 8);
          nig: gen(opr, 0, 9);
          mnr: gen(opr, 0, 10);
          mai: gen(opr, 0, 11);
          myr: gen(opr, 0, 12);
          mei: gen(opr, 0, 13);
        end
      end
    end
  end {condicion};

```

Programa 5.6. (Continuación)

```

begin {instruccion}
  if sym = ident then
    begin i := posicion(id);
      if i = 0 then error (11) else
        if tabla[i].tipo  $\neq$  variable then
          begin {el objeto de la asignacion no es una variable} error (12); i := 0
        end;
      obtsym; if sym = asignacion then obtsym else error (13);
      expresion(symsig);
      if i  $\neq$  0 then
        with tabla[i] do gen(alm, niv-nivel, dir)
      end else
        if sym = callsym then
          begin obtsym;
            if sym  $\neq$  ident then error (14) else
              begin i := posicion(id);
                if i = 0 then error (11) else
                  with tabla[i] do
                    if tipo = procedimiento then gen (lla, niv-nivel, dir)
                    else error (15);
                  obtsym
                end
              end else
                if sym = ifsym then
                  begin obtsym; condicion([thensym, dosym] + symsig);
                    if sym = thensym then obtsym else error (16);
                    ic1 := ic; gen(sac, 0, 0);
                    instruccion(symsig); codigo[ic1].di := ic
                  end else
                    if sym = beginsym then
                      begin obtsym; instruccion([puntoycoma, endsym] + symsig);
                        while sym in [puntoycoma] + syminiinst do
                          begin
                            if sym = puntoycoma then obtsym else error (10);
                            instruccion([puntoycoma, endsym] + symsig)
                          end;
                        if sym = endsym then obtsym else error (17)
                      end else
                        if sym = whilesym then
                          begin ic1 := ic; obtsym; condicion([dosym] + symsig);
                            ic2 := ic; gen(sac, 0, 0);
                            if sym = dosym then obtsym else error (18);
                            instruccion(symsig); gen(sal, 0, ic1); codigo[ic2].di := ic
                          end;

```

Programa 5.6. (Continuación)

```

test(symsig, [ ], 19)
end {instruccion};

begin {bloque} idat := 3; it0 := it; tabla[it] .dir := ic; gen(sal, 0, 0);
if niv > maxniv then error (32);
repeat
  if sym = constsym then
    begin obtsym;
      repeat declaracionconst;
        while sym = coma do
          begin obtsym; declaracionconst
            end;
        if sym = puntoycoma then obtsym else error (5)
        until sym ≠ ident
      end;
    if sym = varsym then
      begin obtsym;
        repeat declaracionvar;
          while sym = coma do
            begin obtsym; declaracionvar
              end;
            if sym = puntoycoma then obtsym else error (5)
            until sym ≠ ident;
      end;
    end;
  while sym = procsym do
    begin obtsym;
      if sym = ident then
        begin poner(procedimiento); obtsym
        end
      else error (4);
      if sym = puntoycoma then obtsym else error (5);
      bloque(niv + 1, it, [puntoycoma] + symsig);
      if sym = puntoycoma then
        begin obtsym; test(syminiinst + [ident, procsym], symsig, 6)
        end
      else error (5)
    end;
    test(syminiinst + [ident], syminidecl, 7)
  until ¬(sym in syminidecl);
  codigo[tabla[it0] .dir] .di := ic;
  with tabla[it0] do
    begin dir := ic; {direccion donde comienza el codigo}
    end;

```

Programa 5.6. (Continuación)

```

ic0 := ic; gen(ins, 0, idat);
instruccion([puntoycoma, endsym] + symsig);
gen(opr, 0, 0); {retornar}
test(symsig, [ ], 8);
listarcodigo;
end {bloque};
procedure interpretar;
  const longpila = 500;
  var d, b, s: integer {registros de direccion del programa, base y parte superior
                       de la pila}
  i: instruccion; {registro de instrucciones}
  p: array [1 .. longpila] of integer; {memoria de datos}
  function base(ni: integer): integer;
  var b1: integer;
  begin b1 := b; {encontrar la base n niveles mas abajo}
    while ni > 0 do
      begin b1 := p[b1]; ni := ni - 1
      end;
    base := b1
  end {base};

begin writeln('EMPIEZA LA EJECUCION PL/0');
  s := 0; b := 1; d := 0;
  p[1] := 0; p[2] := 0; p[3] := 0;
  repeat i := codigo[d]; d := d + 1;
    with i do
      case f of
        lit: begin s := s + 1; p[s] := di
        end;
        opr: case di of {operador}
          0: begin {retornar}
            s := b - 1; d := p[s + 3]; b := p[s + 2];
            end;
          1: p[s] := -p[s];
          2: begin s := s - 1; p[s] := p[s] + p[s + 1]
            end;
          3: begin s := s - 1; p[s] := p[s] - p[s + 1]
            end;
          4: begin s := s - 1; p[s] := p[s] * p[s + 1]
            end;
          5: begin s := s - 1; p[s] := p[s] div p[s + 1]
            end;
          6: p[s] := ord(odd(p[s]));

```

Programa 5.6. (Continuación)

```

8: begin s := s - 1; p[s] := ord(p[s] = p[s + 1])
   end;
9: begin s := s - 1; p[s] := ord(p[s] ≠ p[s + 1])
   end;
10: begin s := s - 1; p[s] := ord(p[s] < p[s + 1])
    end;
11: begin s := s - 1; p[s] := ord(p[s] ≥ p[s + 1])
    end;
12: begin s := s - 1; p[s] := ord(p[s] > p[s + 1])
    end;
13: begin s := s - 1; p[s] := ord(p[s] ≤ p[s + 1])
    end;
end;

car: begin s := s + 1; p[s] := p[base(ni) + di]
   end;
alm: begin p[base(ni) + di] := p[s]; writeln(p[s]); s := s - 1
   end;
lla: begin {generar nueva marca de bloque}
      p[s + 1] := base(ni); p[s + 2] := b; p[s + 3] := d;
      b := s + 1; d := di
   end;
ins: s := s + di;
sal: d := di;
sac: begin if p[s] = 0 then d := di; s := s - 1
   end;
   end {with, case}
until d = 0;
write('ACABA LA EJECUCION PL/0');
end {interpretar};

begin {programa principal}
  for ch := 'A' to ';' do syms[ch] := nulo;
  palabra[1] := 'BEGIN'; palabra[2] := 'CALL';
  palabra[3] := 'CONST'; palabra[4] := 'DO';
  palabra[5] := 'END'; palabra[6] := 'IF';
  palabra[7] := 'ODD'; palabra[8] := 'PROCEDURE';
  palabra[9] := 'THEN'; palabra[10] := 'VAR';
  palabra[11] := 'WHILE';
  symp[1] := beginsym; symp[2] := callsym;
  symp[3] := constsym; symp[4] := dosym;
  symp[5] := endsym; symp[6] := ifsym;
  symp[7] := oddsym; symp[8] := procsym;
  symp[9] := thensym; symp[10] := varsym;

```

Programa 5.6. (Continuación)

```

symp[11] := whilesym;
syms['+'] := mas;
syms['*'] := por;
syms['('] := paren;
syms[')'] := parenc;
syms['='] := igl;
syms['.'] := punto;
syms['<'] := mnr;
syms['≤'] := mei;
syms[';'] := puntoycoma;
nemotecnico[lit] := 'LIT'; nemotecnico[opr] := 'OPR';
nemotecnico[car] := 'CAR'; nemotecnico[alm] := 'ALM';
nemotecnico[lla] := 'LLA'; nemotecnico[ins] := 'INS';
nemotecnico[sal] := 'SAL'; nemotecnico[sac] := 'SAC';
syminidecl := [constsym, varsym, procsym];
syminiinst := [beginsym, callsym, ifsym, whilesym];
syminifac := [ident, numero, paren];
page(output); err := 0;
cc := 0; ic := 0; ll := 0; ch := ''; kk := al; obtsym;
bloque(0, 0, [punto] + syminidecl + syminiinst);
if sym ≠ punto then error (9);
if err = 0 then interpretar else write('ERRORES EN EL PROGRAMA PL/0');
99: writeln
end.

```

Programa 5.6. (Continuación)

E J E R C I C I O S

5.1. Considérese la siguiente sintaxis

$$\begin{aligned}
 I &::= A \\
 A &::= B \mid \text{if } A \text{ then } A \text{ else } A \\
 B &::= C \mid B + C \mid +C \\
 C &::= D \mid C * D \mid *D \\
 D &::= x \mid (A) \mid \neg D
 \end{aligned}$$

¿Qué símbolos son terminales y qué símbolos son no terminales? Determinar los conjuntos de símbolos iniciales, $L(X)$, y siguientes, $F(X)$ de cada símbolo no terminal X . Construir una secuencia de pasos de análisis sintáctico para las frases siguientes:

$$\begin{aligned}
 &x + x \\
 &(x + x) * (+ - x) \\
 &(x * - + x) \\
 &\text{if } x + x \text{ then } x * x \text{ else } - x
 \end{aligned}$$

if x then if $-x$ then x else $x + x$ else $x * x$
 if $-x$ then x else if $x + x$ else x

- 5.2. ¿Satisface la gramática del Ejercicio 5.1 las reglas de restricción 1 y 2, necesarias para el análisis sintáctico con un símbolo delante? Si no las cumple, encontrar una gramática equivalente que si las cumpla. Representar esta sintaxis por medio de un grafo sintáctico, y una estructura de datos que sirva para el Programa 5.3.
- 5.3. Repetir el Ejercicio 5.2 para la siguiente sintaxis:

$$\begin{aligned} I &::= A \\ A &::= B \mid \text{if } C \text{ then } A \mid \text{if } C \text{ then } A \text{ else } A \\ B &::= D = C \\ C &::= \text{if } C \text{ then } C \text{ else } C \mid D \end{aligned}$$

Nota: Puede resultar necesario sustituir o anular algún símbolo no terminal para que sea posible el análisis sintáctico con un símbolo delante.

- 5.4. Estudiar cómo realizar análisis sintáctico descendente con la sintaxis siguiente:

$$\begin{aligned} I &::= A \\ A &::= B + A \mid DC \\ B &::= D \mid D * B \\ D &::= x \mid (C) \\ C &::= +x \mid -x \end{aligned}$$

¿Cuántos símbolos delante, como máximo, se necesitan para poder analizar frases de esta gramática?

- 5.5. Transformar la descripción de PL/0 (Fig. 5.4) en un conjunto equivalente de producciones BNF.
- 5.6. Escribir un programa que determine los conjuntos de símbolos iniciales y sucesores, $I(S)$ y $F(S)$, de cada símbolo no terminal S , de un conjunto de producciones dado.
- Nota:* Utilizar parte del Programa 5.3 para construir una representación interna de la sintaxis, en forma de una estructura de datos, y operar con esta estructura enlazada de datos.

- 5.7. Ampliar el lenguaje PL/0 y su compilador con:

(a) Una instrucción condicional con la forma:

$\langle \text{instrucion} \rangle ::= \text{if } \langle \text{condicion} \rangle \text{ then } \langle \text{instrucion} \rangle \text{ else } \langle \text{instrucion} \rangle$

(b) Una instrucción repetitiva con la forma:

$\langle \text{instrucion} \rangle ::= \text{repeat } \langle \text{instrucion} \rangle \{ ; \langle \text{instrucion} \rangle \} \text{ until } \langle \text{condicion} \rangle$

¿Se presentan dificultades especiales que puedan obligar a cambiar la forma, o la interpretación, de estas ampliaciones al PL/0? No han de introducirse nuevas instrucciones para la máquina PL/0.

- 5.8. Ampliar el lenguaje PL/0 y su compilador con parámetros de procedimiento. Estudiar dos posibles alternativas e implementar una de ellas.
- (a) *Parámetros valor.* Los parámetros reales en la llamada son expresiones cuyos valores se asignan a variables locales, representadas éstas por los parámetros formales especificados en la cabecera del procedimiento.
- (b) *Parámetros variable.* Los parámetros reales son variables y, al realizarse una llamada al procedimiento, sustituyen a los parámetros formales. Los parámetros variable se implementan pasando al procedimiento la dirección del parámetro real, y almacenándola en el lugar designado por el parámetro formal. A continuación, el parámetro variable se accede indirectamente por medio de su dirección. Por lo tanto, los parámetros variable proporcionan un mecanismo para acceder a variables definidas fuera de los procedimientos, y las reglas de ámbito de variables pueden modificarse como sigue: Dentro de un procedimiento, sólo se puede acceder directamente a variables locales; el acceso a variables no locales sólo puede hacerse a través de parámetros.
- 5.9. Ampliar el lenguaje PL/0 y su compilador con arrays de variables. Supóngase que el rango de índices de una variable array a se indica en su declaración como:

var a (desde: hasta)

- 5.10. Modificar el compilador PL/0 para que genere código para el computador del que disponga el lector.
- Nota:* Generar código ensamblador simbólico para evitar problemas con el programa cargador («loader»). En una primera etapa, no intentar optimizar el código en relación con, por ejemplo, el uso de registros. Las posibles optimizaciones deberían hacerse en un cuarto paso de refinamiento del compilador.
- 5.11. Ampliar el Programa 5.5 para obtener otro llamado «impresionbonita». El objetivo de este programa será leer textos PL/0, e imprimirllos según un esquema gráfico que refleje de forma natural la estructura del texto del programa, mediante una apropiada separación de líneas y fijación de márgenes a cada línea. Definir primero de manera precisa la separación de líneas y las reglas de fijación de márgenes, de acuerdo con la estructura sintáctica de PL/0. A continuación, implementarlas añadiendo instrucciones de escritura al Programa 5.5. (Por supuesto, se deben suprimir las instrucciones de escritura del analizador léxico.)

R E F E R E N C I A S

- 5-1. AMMANN, U., «The Method of Structured Programming Applied to the Development of a Compiler», *International Computing Symposium 1973*, A. Günther y otros eds., (Amsterdam: North-Holland Publishing Co., 1974), 93-99.
- 5-2. COHEN, D. J. y GORIKAH, C. C., «A List Structure Form of Grammars for Syntactic Analysis», *Comp. Survey*, 2, No. 1 (1970), 65-82.

- 5-3. FLOYD, R. W., «The Syntax of Programming Languages – A Survey», *IEEE Trans.*, EC-13 (1964), 346-53.
- 5-4. GRIES, D., *Compiler Construction for Digital Computers* (New York: Wiley, 1971).
- 5-5. KNUTH, D. E., «Top-down Syntax Analysis», *Acta Informatica*, 1, No. 2 (1971), 79-110.
- 5-6. LEWIS, P. M. y STEARNS, R. E., «Syntax-directed Transduction», *J. ACM*, 15, No. 3 (1968), 465-88.
- 5-7. NAUR, P., ed., «Report on the Algorithmic Language ALGOL 60», *ACM*, 6, No. 1 (1963), 1-17.
- 5-8. SCHORRE, D. V., «META II, A Syntax-oriented Compiler Writing Language», *Proc. ACM Natl. Conf.*, 19, (1964), D 1.3.1-11.
- 5-9. WIRTH, N., «The Design of a PASCAL Compiler», *Software-Practice and Experience*, 1, No. 4 (1971), 309-33.

A EL CONJUNTO DE CARACTERES ASCII

y \ x	0	1	2	3	4	5	6	7
0	nul	dle		0	@	P	'	p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	\$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
8	bs	can	(8	H	X	h	x
9	ht	em)	9	I	Y	i	y
10	lf	sub	*	:	J	Z	j	z
11	vt	esc	+	;	K	[k	{
12	ff	fs	,	<	L	\	l	
13	cr	gs	-	=	M]	m	}
14	so	rs	.	>	N	↑	n	~
15	si	us	/	?	O	-	o	del

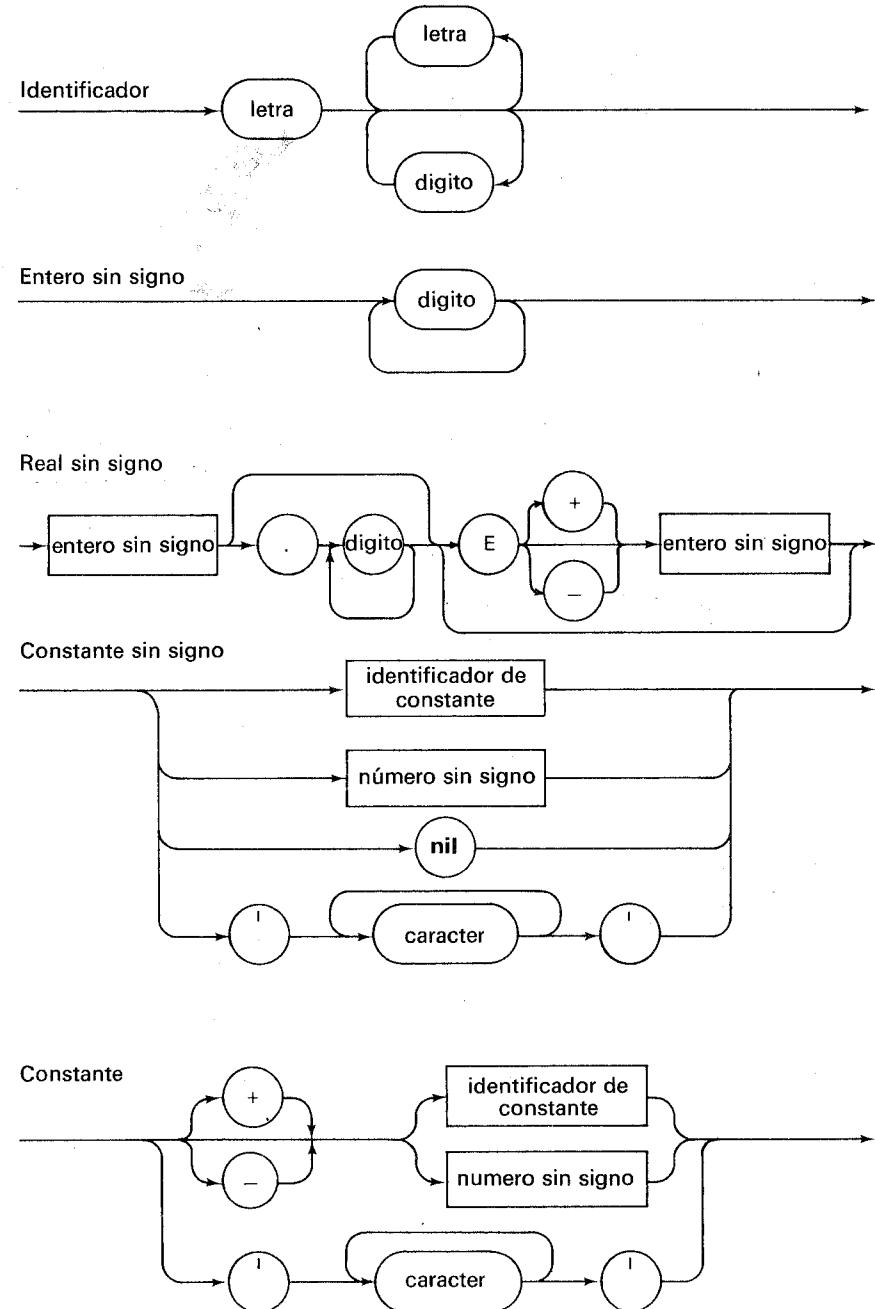
El número de orden de un carácter, ch , se calcula a partir de sus coordenadas en la tabla mediante la fórmula

$$ord(ch) = 16*x + y$$

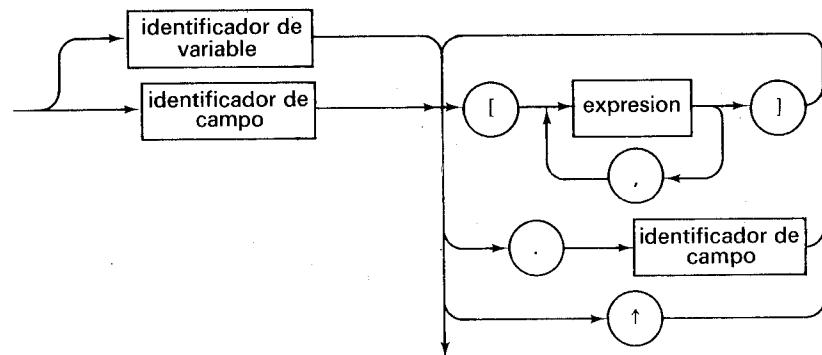
Los caracteres con número de orden de 0 a 31 y número 127 se denominan *caracteres de control* y se utilizan para transmisión de datos y control de dispositivos. El carácter número 32 es el blanco.

B

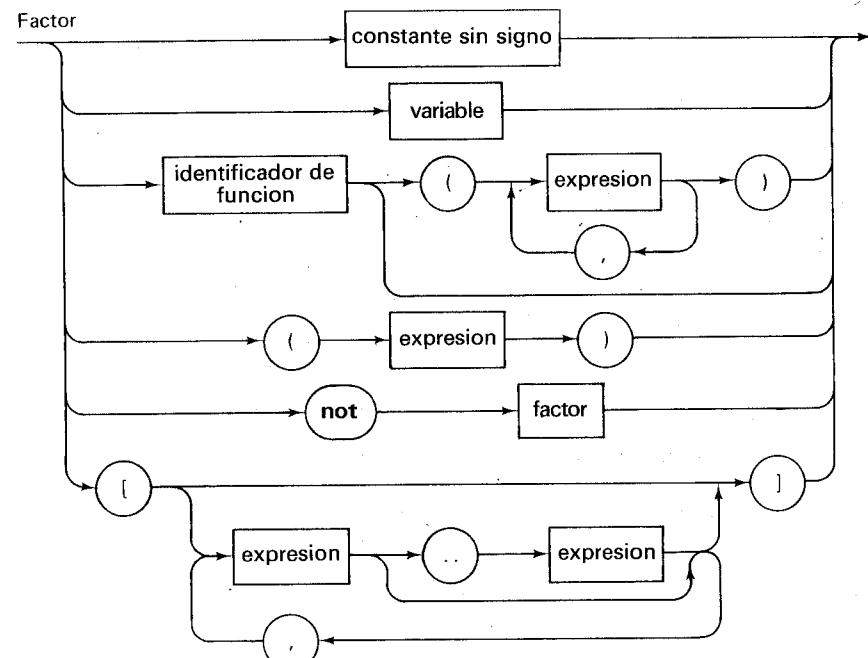
DIAGRAMAS SINTACTICOS DE PASCAL



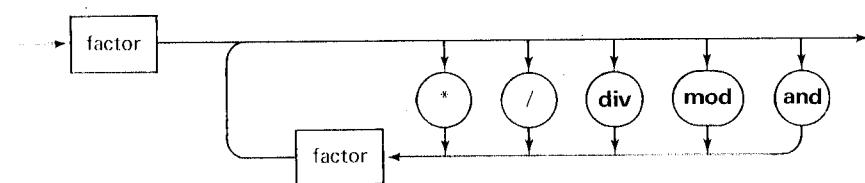
Variable



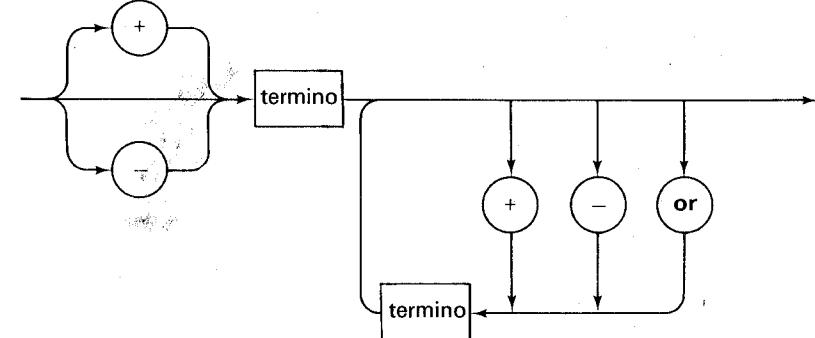
Factor



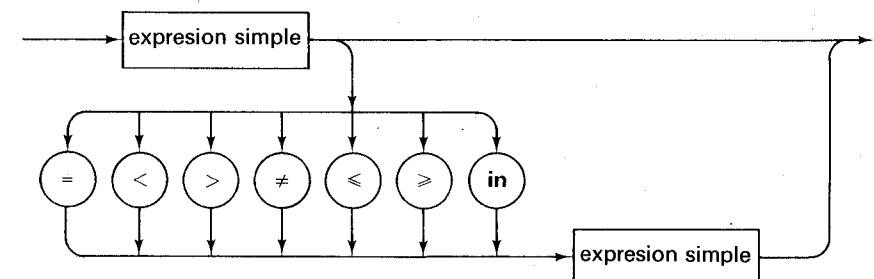
Término



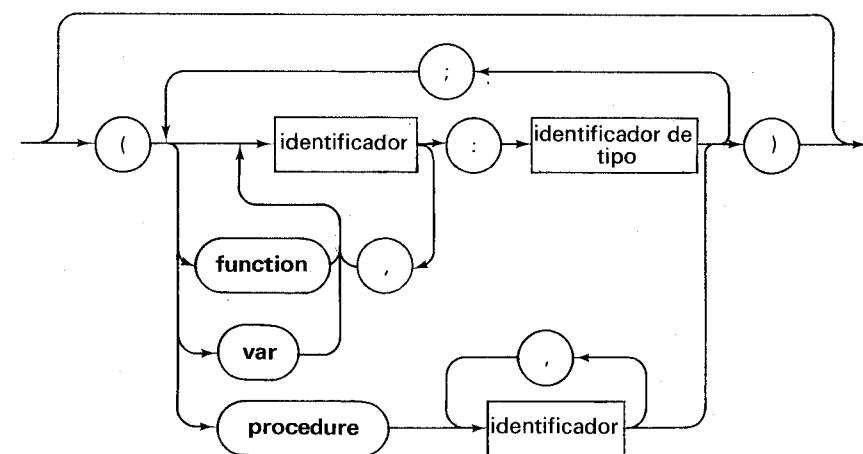
Expresión simple

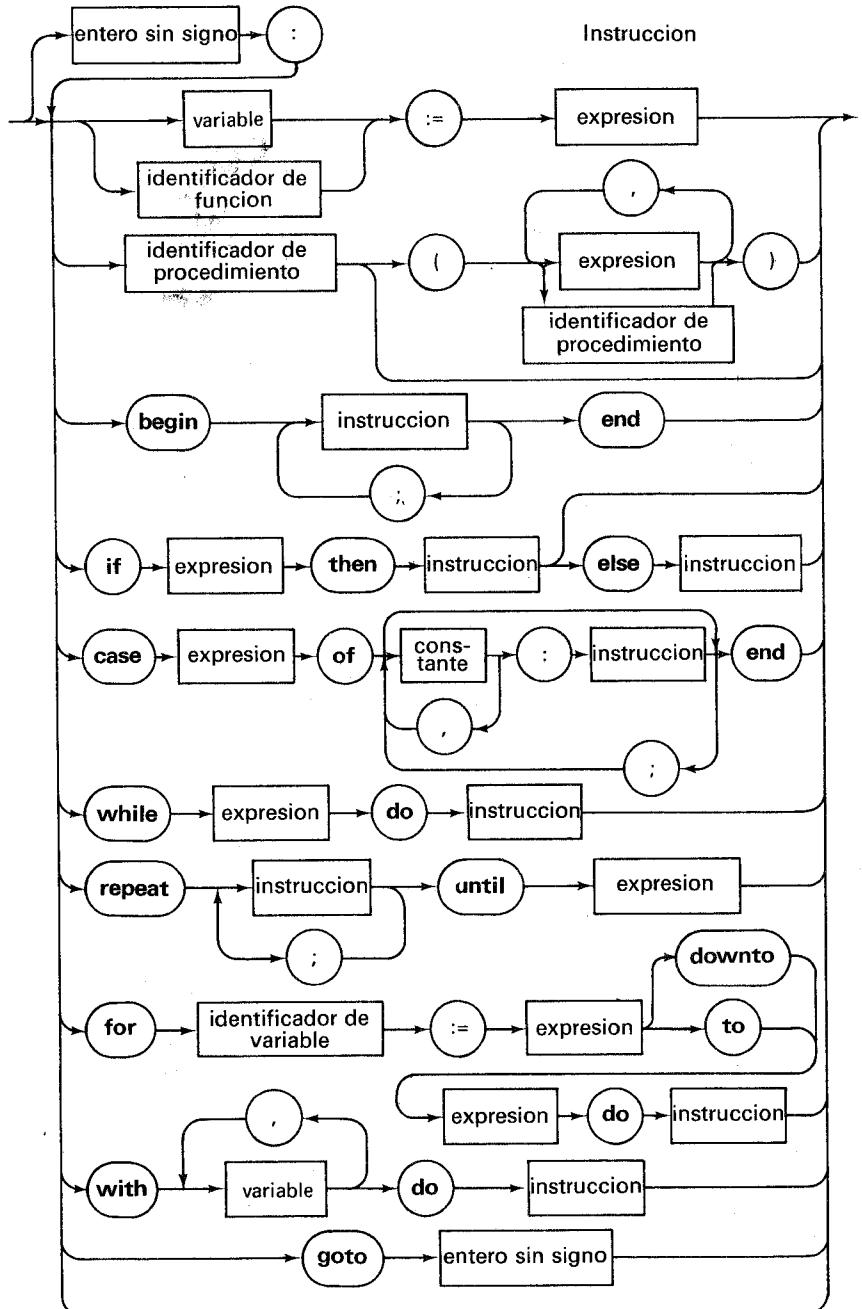
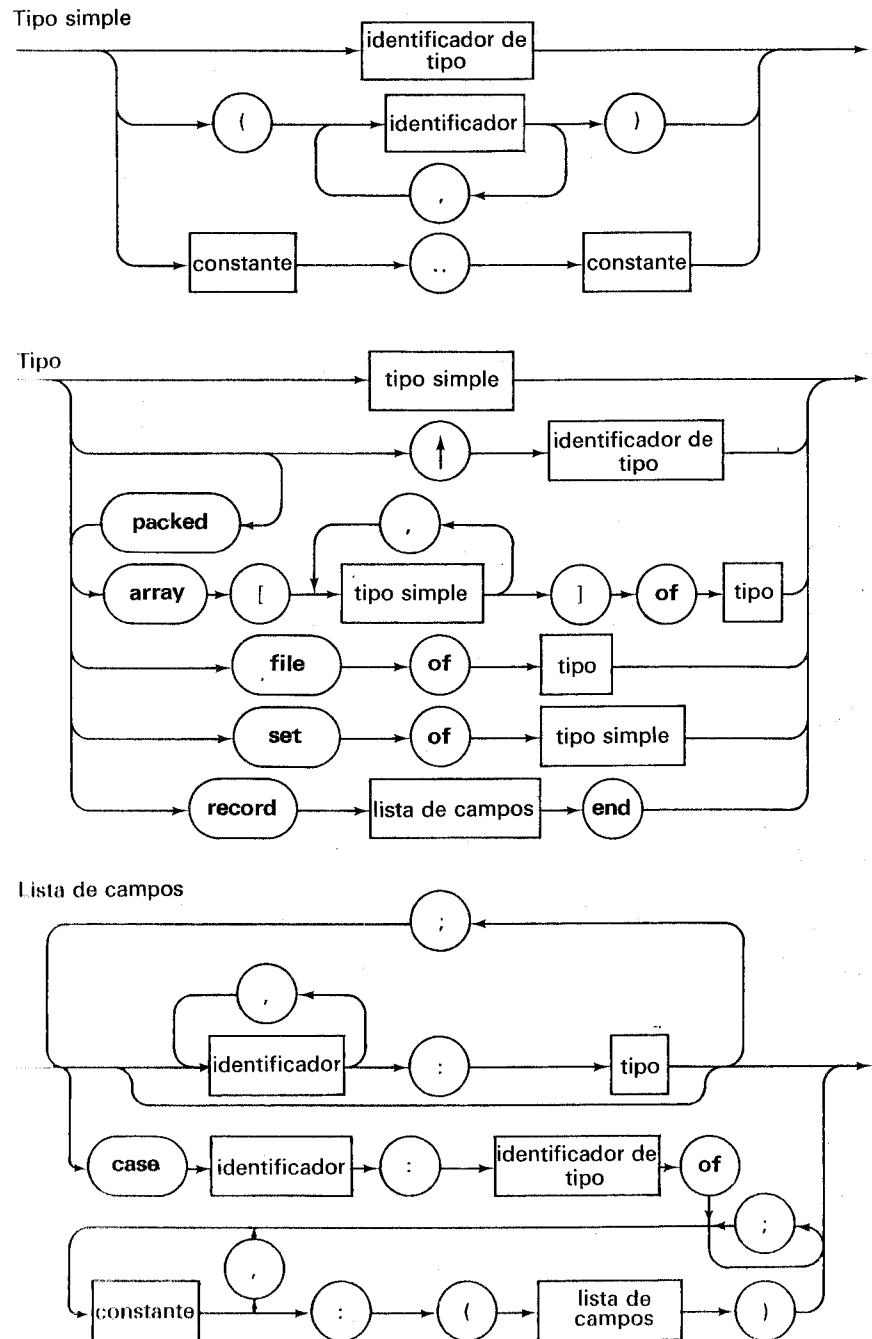


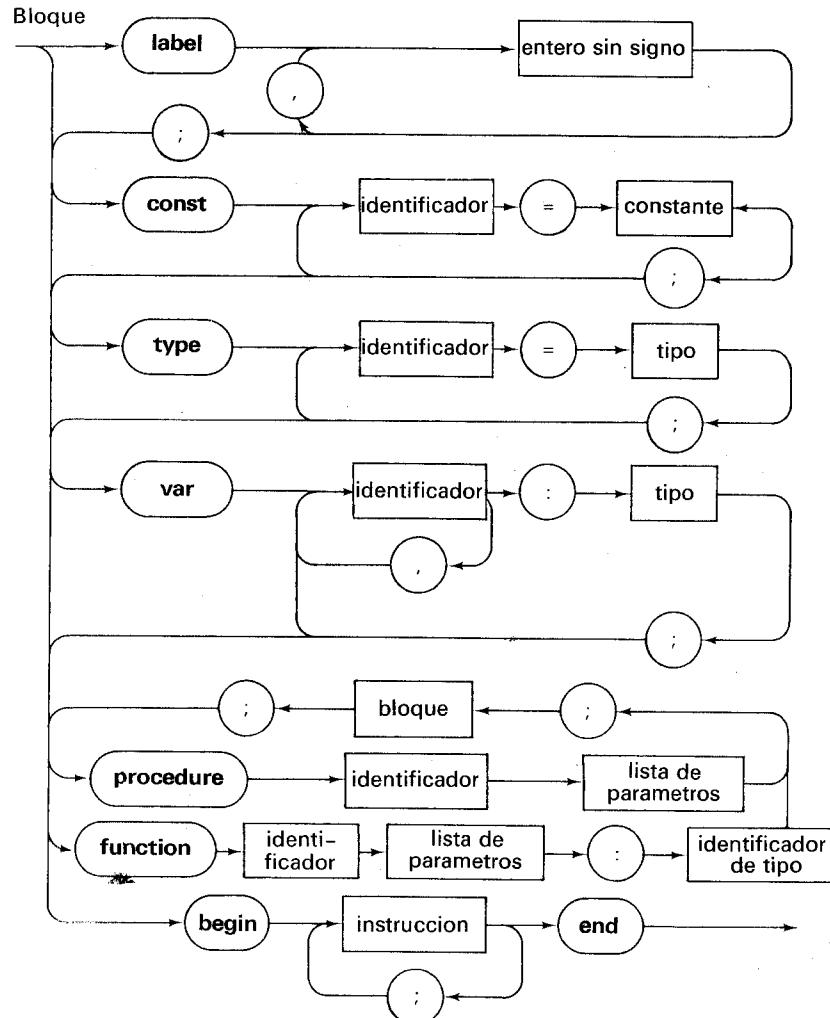
Expresión



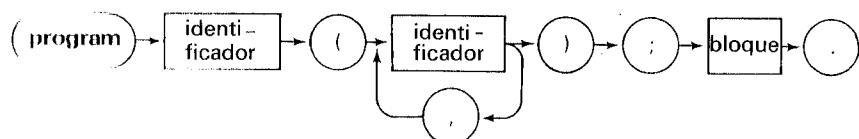
Lista de parámetros







Programa



INDICE DE MATERIAS

- Abstracción, 1
- Acceso aleatorio, 12
- Actualización selectiva, 15, 180
- Adelson Velskii, G. M., 229
- Agruparse, 284
- Ajuste, 32
- alfa (type), 34
- ALGOL 60, 4, 12
- Ambito, 326
- Análisis por objetivos, 306
- Análisis sintáctico, 300
- Analizador lexicográfico («Scanner»), 28
- Antecesor m 204
- «Antipánico», regla, 340
- Arbol, 202
 - altura, 204
 - borrado, 224, 236, 267
 - búsqueda, 215, 231, 264, 274, 275
 - grado, 204
 - inserción, 215, 231, 264, 274, 275
 - nivel, 204
 - profundidad, 204
 - recorrido, 212
- Arbol B binario, 273
- Arbol BB, 273
- Arbol BBS, 275
- Arbol de página, 261
- Arbol 2-3, 273
- Arbol Fibonacci, 230
- Arboles B, 261
- Arboles equilibrados, 229
- Arboles, hashing con, 295
- Arboles multicamino, 258
- Area de desbordamiento, 283
- Armónico, número, 70, 228, 290
- Array, selector de, 13
- ASCII, 10, 46
- Asignación dinámica, 177
- AVL, árbol, 229, 280
- «Backus-Naur-Form» (BNF), 298
- Bayer, R., 261, 268, 274
- Binaria, búsqueda, 15, 58
- Binaria, inserción, 66
- Binario, árbol, 206
- boolean, 9
- «Branch and Bound», 168
- Burbuja, método de la, 71
- Búsqueda, árbol de, 213
- Búsqueda binaria, 15, 58
- Camino externo, longitud de, 204
- Camino interno, longitud de, 204
- Campo indicador, 22
- Caracteres de control, 46
- Característica, función, 36
- Cardinalidad, 5, 7, 13, 18, 26
- Carga, factor de, 290
- Cartesiano, producto, 18
- Cascada, mezcla en, 132
- Case, instrucción, 24
- Centinela, 15, 66, 187, 213
- Centroide, 246
- Claves, transformación de, 281
- Cola, 182
- Colisión, 282
- Compartidos, datos, 177
- Compensada, mezcla, 95, (Véase también Equilibrada)
- Concatenación, 38
- Concordancia, 186
- Conjunto potencia, 25
- Constructor, 7, 13, 19
- Contexto, dependencia del, 332
- Contexto, libre de, 299
- Contexto, sensible al, 300
- Control, caracteres de, 46
- Convencional, notación, 212
- Corrutina, 127
- Cuadrática, inspección, 285
- Curva de Hilbert, 140
- char*, 10
- chr(x)*, 11, 17
- Declaración de tipo, 4, 12, 18, 23, 39, 178
- Delante, 301
- Dependencia, diagrama de, 332
- Descendente, análisis, 301
- Descendiente, 202
- Desplazamiento, 35
- Diagrama de dependencia, 332
- Diferencia de conjuntos, 26
- Dijkstra, E. W., xii
- Direccion, 32
- Direccionamiento vacío, 284
- Dirigido por tabla, 306
- Discriminante de tipo, 12