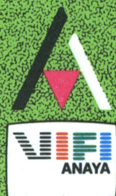
Peter Brown

PASCAL a partir del BASIC



Contiene diagramas de sintaxis de PASCAL y TURBO PASCAL

PASCAL A PARTIR DEL BASIC

Alejandro Sánchez Valderrábano

PASCAL a partir del BASIC

Peter Brown

MICROINFORMATICA

Título de la obra original:
PASCAL FROM BASIC

Traducción de: Santiago García Conde
Diseño de colección: Antonio Lax
Diseño de cubierta: Narcís Fernández

Primera edición, 1985.
Primera reimpresión, 1986.

Reservados todos los derechos. Ni la totalidad ni parte de este libro puede reproducirse o transmitirse por ningún procedimiento electrónico o mecánico, incluyendo fotocopia, grabación magnética o cualquier almacenamiento de información y sistema de recuperación, sin permiso escrito de Ediciones Anaya-Multimedia, S.A.

Traducción autorizada de la edición en inglés © 1982, Addison-Wesley Publishing Company, Inc. Esta traducción es publicada y vendida con permiso de Addison-Wesley Publishing Company, Inc., propietaria de todos los derechos de edición y venta de la misma.

© EDICIONES ANAYA MULTIMEDIA, S. A., 1986
Villafranca, 22. 28028 Madrid
Depósito legal: M. 6.676-1986
ISBN: 84-7614-012-6
Printed in Spain
Imprime: Anzos, S. A. Fuenlabrada (Madrid)

Indice

1.—Un ejemplo para enseñar los fundamentos 17

Un ejemplo 17
Declaraciones 19
El programa 21
Sentencias individuales 23
Números de línea 25
Sumario 25
Algunos amigos 26

2.—Los objetivos del PASCAL 29

Historia 29
Características específicas de diseño 30
Definición del PASCAL 31
Las variaciones en el BASIC y en el PASCAL 31
Portabilidad 32
Método de ejecución 32
Algunas consideraciones prácticas 33

Puntos o cuestiones relativos a los lenguajes 34
Mantenimiento 34
Niveles de legibilidad 35
Hacer que las sentencias sean fáciles de leer 36
Estructuración 37
Relación con el problema 37
Imponer una disciplina 38
Separación de las partes que puedan necesitar cambios 39
Errores 39
Evaluación 40
Los seres humanos y la disciplina 41
El cambio en la forma de pensar 42

3.—Sistemas operativos y editores 45

Los microordenadores y los grandes ordenadores 45
Lenguajes interactivos y no interactivos 46
Preparación de un programa PASCAL 47
Sistemas operativos 48
Sistemas de archivo 49
Lenguajes de mando (*command languages*) 50
Niveles de comunicación 51
Puntos varios 52
Editores 53
Rasgos o características de los editores 54
Ordenes o comandos de edición 55
Displays o sistemas de presentación visual perfeccionados 56
Utilización de un compilador de PASCAL 57
Ejecución de un programa PASCAL 58
Depuradores o *debuggers* 58
Casos especiales 59
La opinión de un experto 60

4.—Traducción de conceptos del BASIC 63

El espaciado y los comentarios u observaciones 64
Nombres 64
Tipos de datos 65
Tipos subrango 66
Declaraciones 68
Funciones incorporadas 68
Constantes 69
Expresiones 70
Sentencias IF 71

Saltos hacia atrás.. 73
Sentencias GOTO.. 73
Sentencias FOR.. 74
Sonidos y gráficos.. 77
Las sentencias ON.. 77
Sentencias BASIC sin equivalencias en PASCAL............. 78
Constantes, tipos y declaraciones........................... 79

5.—Subrutinas y funciones.. 83

Conceptos básicos.. 83
Argumentos y parámetros.. 85
Puntos a observar.. 86
Cómo trabaja la función.. 88
Procedimientos.. 89
Procedimientos incorporados.. 90
Declaraciones locales.. 90
Refinamiento progresivo.. 91
El ámbito local.. 92
Espacio de memoria para las variables........................ 94
Diferencias con respecto a la asignación permanente de memoria 96
La adaptación a la estructura en bloques.................... 96
Variables no locales.. 97
Repetición o recurrencia.. 98
Las variables locales y la repetición........................ 100
La referencia hacia adelante.. 102
Cambio del valor de los argumentos........................ 103
Las rutinas como parámetros.. 104
Construyendo bloques.. 104
Las herramientas.. 105

6.—Más sobre tipos sencillos de datos........................ 109

Clases de tipos.. 109
Datos Booleanos.. 110
Datos de carácter.. 112
Operaciones con los tipos sencillos........................ 113
Juegos de valores ordenados.. 114
El control de los bucles.. 116

7.—Matrices (*arrays*) y cadenas (*strings*)........................ 119

Conceptos básicos.. 119
Bucles con matrices.. 121

Un aparte acerca de la detección de errores 122
Declaración de los tipos de índice 122
Sentencias MAT 125
Matrices de matrices 126
Un ejemplo 127
Malas noticias 129
Un ejemplo de parámetro matricial 130
Matrices compactadas (*Packed arrays*) 131
Los *strings* o cadenas en BASIC 132
Los *strings* o cadenas en PASCAL 133
Superando las restricciones 135

8.—Registros 139

La declaración de un registro 139
Tipos estructurados 140
Estructuración de datos 141
Facilidades para el anidamiento 143
Una opinión alternativa 144
Un ejemplo 144
Abstracciones 146
La sentencia *with* (con) 147
Registros con variantes 149
Ventajas de las variantes 151
Una forma alternativa de las variantes 152
Equivalencia de nombre 153

9.—Entrada y salida 155

Categorías de archivos 155
Los tipos de datos en entrada/salida 157
Nombres de archivos externos 159
Correspondencia entre archivos externos y reales 160
Características de los archivos en PASCAL 161
Los procedimientos *get* y *put* 162
Inicialización de archivos 164
La función *eof* 164
Los archivos como parámetros 166
Operaciones sobre archivos 167
Archivos y matrices 168
Propiedades especiales de los archivos de texto 168
Los procedimientos *writeln* y *readln* 170
El final-de-archivo en los archivos de texto 172
Entrada/salida interactivas 173

Comprobación interactiva del fin-de-archivo, o *eof* 176
Formatos de salida 177
Longitud de la parte fraccionaria 178
Otros puntos 179
Tipos de datos encarcelados o recluidos 180
Gráficos 182
Sumario 182

10.—Conjuntos 185

Introducción a los conjuntos 185
Un ejemplo de conjunto 186
Operaciones con conjuntos 187
Operaciones de relación con conjuntos 189
Expresiones dentro de constructores de conjuntos 190
Un ejemplo completo con caracteres 190
Un segundo ejemplo completo 191
Conjuntos desemejantes 192
Restricciones en relación con los conjuntos 192
Otras operaciones con conjuntos 193

11.—Almacenamiento o memoria dinámica 197

Deficiencias de la memoria estructurada en bloques 197
Un problema análogo 199
La pila 199
La utilización de la memoria dinámica 200
Punteros 200
Los tipos de datos del almacenamiento dinámico 201
Toma en préstamo y devolución o liberación 202
Referencias a las variables dinámicas 203
Sumario intermedio 205
Variables dinámicas enlazadas (*linked*) 206
Estilo de programación 208
Ejemplo de procedimientos de tratamientos de listas 209
La impresión de una lista 210
Adición de elementos a una lista 211
Para borrar o suprimir (*delete*) de una lista 214
Punteros adicionales 217
Sumario sobre los usos del almacenamiento dinámico 218

12.—Bibliotecas 221

Bibliotecas o códigos fuente y objeto 221
Bibliotecas de códigos objeto con seguridad 223

Bibliotecas en otros lenguajes 224
La sentencia CHAIN 224
Bibliotecas en código fuente 225

13.—Resumiendo 227

El saldo 227
La adaptación del estilo 228
El esfuerzo en la preparación de programas 230
La ampliación o extensión 231
Palabras finales 231

Referencias 233

Apéndice A.—Procedimientos y funciones incorporados 235

Procedimientos 236
Funciones 236

Apéndice B.—Sumario del PASCAL 238

Apéndice C.—Diagramas de sintaxis 243

Explicación 243
Limitaciones 245
Diagramas de sintaxis del PASCAL estándar 246
Diagramas de sintaxis del TURBO PASCAL 251

A mi esposa, que me ha importunado para que terminara;
mi amiga, que me ha levantado el ánimo;
mi colega, que me ha criticado;
mi jefe, que me entiende de sobra.

A la persona única en que se reúnen todas éstas.

EL FORASTERO: Por favor, ¿por dónde puedo ir a Correos?
EL NATIVO: Si yo tuviera que ir a Correos, no empezaría desde aquí.

Prefacio

Este libro parte de la suposición de que sus lectores son programadores razonablemente competentes en BASIC, y que desean aprender PASCAL. La mayoría de los libros de PASCAL parecen escritos para lectores totalmente carentes de cualquier noción de programación. Sin embargo, tu problema es distinto del de un novato. No necesitas que se te expliquen conceptos básicos, como los de variables, bucles, etc.; lo que necesitas es que se te ayude par adaptar tu forma de pensar desde el BASIC al PASCAL. En realidad, la tarea a la que te enfrentas es probablemente más ardua que la del novato, porque el BASIC es en realidad un elemento heterodoxo e insolidario entre todos los lenguajes de programación... aunque, a pesar de ello (o, tal vez, por ello mismo), se ha convertido en el lenguaje más popular del mundo. Es más difícil aprender unos conceptos, y luego volver a aprenderlos de nuevo en una forma diferente, que empezar completamente desde cero. No lo dudes, el paso desde el BASIC al PASCAL supone un cambio drástico. Si escribes tus programas en PASCAL en la misma forma que tus programas en BASIC, te parecerás a un turista inglés en Francia traduciendo sus frases, palabra por palabra, con la ayuda de un diccionario.

Lo que este libro pretende específicamente es ayudarte a superar los problemas que los programadores en BASIC encuentran en el PASCAL, y a realizar la consiguiente adaptación en tu estilo de programar. No suponemos que

vayas a decir adiós para siempre a tu viejo amigo el BASIC. En realidad, nuestro libro se muestra bastante crítico acerca de determinados aspectos del PASCAL, y trata de exponer claramente al lector aquellas cosas en las que el PASCAL gana, y las otras en las que pierde.

Tratamos también de mantener en esta obra un tono ligero, que haga su lectura divertida. Nuestro objetivo es la seriedad, pero sin solemnidad ni pesadez. Los conceptos se van introduciendo casi exclusivamente con ayuda de ejemplos que, cuando ello conviene, se relacionan o comparan con el BASIC. Si prefieres un enfoque o un tratamiento más formal o estirado, busca en otra parte.

Cuando hayas leído el libro, deberás haber adquirido dos capacidades. Deberás ser capaz de escribir buenos programas en PASCAL, y también de leer libros o manuales referentes a dicho lenguaje y entender lo que tratan.

Este libro mantiene una exquisita neutralidad respecto a las distintas versiones de PASCAL que compiten entre sí: con una honradez absoluta, no menciona a ninguna de ellas. Afortunadamente, los distintos sistemas de PASCAL se parecen mucho más entre sí que los distintos sistemas de BASIC, de modo que es posible dar una descripción detallada sin que ello implique comprometerse con una implementación concreta o determinada. Así, por ejemplo, importa poco si piensas usar el PASCAL en un ordenador personal o en una gran unidad central utilizada en tiempo compartido.

Reconocimientos

Me gustaría expresar mi gratitud a Laurence Atkinson, que, con entusiasmo, me inspiró, sin darse cuenta de ello, la idea de escribir este libro. También he de agradecer a Tony Hoare su valiosa orientación, así como a varios de mis colegas, incluyendo a Stephen Binns, David Turner y, muy especialmente, a Peter Welch. Un simpático y bondadoso sistema PASCAL se ha ocupado de la composición de los programas mostrados en este libro.

Finalmente, permítaseme dejar constancia de mi agradecimiento a dos grandes personas: Marianne Kong, que mecanografió el manuscrito con un cuidado y una meticulosidad realmente maravillosos y soportó con enorme paciencia mis cambios, y Heather Brown, que empleó tantas horas en la revisión y la crítica del libro como yo en escribirlo. Sus cáusticos comentarios y sus violentas condenas consiguieron aniquilar las partes peores del libro antes de que llegaran a la imprenta.

PETER BROWN

Canterbury, octubre 1981.

Comienza en el comienzo.

DYLAN THOMAS

1 Un ejemplo para enseñar los fundamentos

Un ejemplo

Para ir haciéndonos una idea de la tarea que tenemos ante nosotros, empezaremos con un programa BASIC muy sencillo y mostraremos su equivalente en PASCAL. EL programa toma como datos un número N, al que siguen otros N números. Todo lo que realiza el programa es imprimir la media aritmética de los mismos. En BASIC, este programa se puede escribir así:

```
10   REM———HALLAR LA MEDIA ARITMETICA
     DE N NUMEROS———
100  INPUT N
110  LET S = 0
120  FOR K = 1 TO N
130     INPUT X
140     LET S = S + X
150  NEXT K
200  PRINT "MEDIA =";S/N
999  END
```

Una traducción directa de esto en PASCAL —que, como veremos, no es un buen programa PASCAL— es la siguiente:

```
program mediaaritmética(input, output);
(* halla la media aritmética de N números *)
var
    n: integer;
    k: integer;
    s: real;
    x: real;
begin
    read(n);
    s:= 0;
    for k:=1 to n do
    begin
        read(x);
        s:= s + x;
    end;
    writeln ('Mediaaritmética =', s/n);
end.
```

Si no habías visto hasta ahora un programa PASCAL, te llamarán inmediatamente la atención muchas diferencias con respecto al BASIC.

La más patente es, paradójicamente, una de las menos importantes: el hecho de que el programa en BASIC emplee letras mayúsculas, mientras que el PASCAL utiliza minúsculas. Lo primero que hay que decir es que se trata en gran medida de una cuestión de costumbre, no de necesidad. Podríamos haber escrito el programa BASIC en minúsculas y el PASCAL en mayúsculas, y los programas seguirían siendo aceptables para la mayoría de los compiladores.

El hábito —y no es un hábito universal— de usar letras mayúsculas en BASIC es, en buena medida, resultado de la historia. El BASIC se desarrolló a comienzos de la década de los sesenta, cuando los ordenadores tenían unos juegos de caracteres muy pequeños; las entradas a los ordenadores se hacían frecuentemente desde tarjetas perforadas o máquinas de escribir bastante primitivas, y a menudo no se disponía de letras minúsculas. En épocas más recientes, tanto los ordenadores como los dispositivos desde los que se pueden introducir datos en ellos han sido dotados de juegos de caracteres más ricos. Ahora ya no tenemos que GRITAR AL ORDENADOR EN LETRAS MAYUSCULAS, sino que podemos comunicarnos en minúsculas de un modo mucho más refinado. La oscilación del péndulo ha llegado ahora tan lejos que muchos programadores evitan totalmente el uso de las mayúsculas, y los manuales de programación para los lenguajes más recientes tienden a hacer lo mismo. Seguramente el péndulo volverá algún día a ocupar una posición más moderada.

No obstante, la convención actual nos resulta bastante útil para este libro. Presentaremos en él los programas de BASIC en mayúsculas y los de PAS-

CAL en minúsculas. Ello nos permitirá hablar de los programas y establecer comparaciones entre ellos sin riesgo alguno de ambigüedad.

Puede verse también que los programas en PASCAL contienen palabras en negrita, por ejemplo **begin**. También esto es un hábito o costumbre de presentación más que una propiedad del lenguaje. Cuando el programa se introduce en el ordenador desde el teclado, las letras negritas se escriben como letras ordinarias.

Declaraciones

Una segunda faceta llamativa de los programas es que en BASIC la acción empieza ya desde la primera línea (o, estrictamente hablando, desde la segunda, ya que la primera es un comentario), mientras que en el programa de PASCAL hay media docena de líneas de introducción antes de que realmente empiece a ocurrir algo. La primera línea es

program *mediaaritmética*(*input, output*);

Al principio de todo programa PASCAL ha de venir una línea de esta forma, en la que aparezca el nombre que elijamos para el programa, en este caso *mediaaritmética*. Más adelante explicaremos esta línea de encabezamiento del programa. La línea que sigue al encabezamiento es un comentario PASCAL, semejante a una sentencia REM en BASIC.

Mucho más importantes son las líneas

```
var
    n: integer;
    k: integer;
    s: real;
    x: real;
```

Estos son ejemplos de *declaraciones*. En BASIC, si queremos usar una variable X podemos hacerlo sin preparativos ni ruido alguno; en PASCAL, en cambio, tenemos que declarar la variable X antes de poder usarla. El BASIC, desde luego, incluye el concepto de las declaraciones. Si en BASIC queremos usar una matriz o *array* (una tabla) Q, tenemos que declararla por medio de una línea tal como

```
DIM Q (5, 8)
```

En el PASCAL existe la misma idea, pero hay que declararlo *todo*.

Cuando declaramos una variable, tenemos que especificar su *tipo de datos.* Esto de los tipos de datos constituye uno de los rasgos más potentes y excitantes del PASCAL, aunque ello no quede de manifiesto en nuestro presente ejemplo de programa. El PASCAL tiene ya incorporados algunos tipos de datos, como los tipos *integer* y *real* utilizados en el ejemplo, y nos permite también definir nuestros propios tipos, como se verá más adelante. En el BASIC existe ya la idea embrionaria del tipo de datos. Las variables son normalmente del tipo de datos *numérico,* como las N, K, S y X en nuestro ejemplo. Sin embargo, si se añade el símbolo de dólares al nombre de una variable, ésta es del tipo de datos de *string,* o cadena de caracteres, por ejemplo, A$, P9$. Algunos lenguajes BASIC emplean también un tipo de datos *integer* o entero, que frecuentemente se representa por nombres tales como K%, T%.

El tipo de datos especifica el juego o conjunto de valores que puede tomar un objeto. Con cada tipo de datos va asociado un conjunto de operadores; por ejemplo, podemos aplicar un operador multiplicador a los datos numéricos. No es posible multiplicar entre sí dos literales o cadenas de caracteres, pero existen otras operaciones que se aplican únicamente a las cadenas, tales como la extracción de una subcadena (*substring*). Algunos operadores son *polimórficos,* en cuanto que se pueden aplicar a más de un tipo de datos. En BASIC, por ejemplo, ocurre esto con el operador de asignación y con algunos operadores en sentencias IF, por ejemplo:

```
LET X = 3
LET X$ = "STRING"

IF X = Y THEN 30
IF X$ = Y$ THEN 50
```

Al PASCAL le son aplicables reglas similares. Los tipos de datos del BASIC se ponen de manifiesto por el nombre de la variable: X es numérica, y X$ es una cadena. En PASCAL se puede utilizar cualquier nombre para cualquier variable. Cuando declaramos la variable damos su nombre y el tipo de datos asociado con ella. Más adelante veremos que en los nombres del PASCAL existe libertad de elección. En lugar del nombre *s* podríamos haber usado (en realidad, deberíamos haberlo hecho) el nombre *suma* o incluso *sumadevariables.* Como principio general, lo mejor es elegir nombres que manifiesten claramente el objeto o finalidad de las variables; si hemos utilizado estos nombres formados por una sola letra, ha sido para mantener una correspondencia directa con el BASIC.

Los tipos de datos utilizados en el programa de *mediaaritmética* son *reales* (*real*), lo que equivale al tipo numérico del BASIC, y *enteros* (*integer*). De las variables *s* y *x* se declara que son variables reales, y de las *n* y *k,* que son variables *enteras* o *integer.* (Podríamos haber escrito estas cuatro declaraciones en cualquier orden, lo mismo que en BASIC las cadenas o *strings* se pueden declarar en cualquier orden.) En PASCAL es importante distinguir aquellas variables que sólo pueden tomar valores enteros, como la *k* y la *n* en el programa de *mediaaritmética,* de aquellas que pueden tomar cualquier valor numérico,

como las *s* y *x*. Las razones para ello se desprenden principalmente del diseño de los ordenadores actuales. Si se sabe que una variable toma solamente valores enteros, es posible almacenarla en una forma distinta y más compacta que la de una variable que pueda adoptar cualquier valor numérico. Además, las operaciones realizadas con enteros se ejecutan a una velocidad mucho mayor que las hechas con valores reales; esta relación de velocidades puede variar desde dos a más de cien. Finalmente, las operaciones con enteros son exactas. Las operaciones con variables reales son inexactas y pueden dar pequeños errores de redondeo. Es muy posible que hayas experimentado los efectos de ese redondeo al ejecutar un programa de BASIC; tal vez esperaras que el resultado de un programa fuera 6 y, en lugar de ello, hayas obtenido como resultado 6,000000001.

En la mayoría de los lenguajes BASIC, se hace caso omiso de la diferencia entre números enteros y reales, haciendo a todos reales (numéricos), y uno se aguanta con las ocasionales excentricidades, tales como la del 6,000000001. En PASCAL algunas veces es posible hacer esto sin que pase nada, pero no se puede usar una variable real en una sentencia **for** o como subíndice de una matriz o *array*. Dadas estas exigencias y sus consecuencias indirectas, lo mejor es distinguir los números enteros de los reales. El único problema que puede surgir cuando se usa el tipo de datos *integer* o enteros es que algunas implementaciones imponen unos límites bastante severos sobre el valor de los enteros. Por ejemplo, pueden estar prohibidos éstos por encima de 32.767. Para más detalles, consulta tu manual particular.

El programa

Al mirar las instrucciones ejecutables del programa PASCAL, te hallarás en un terreno más familiar. Las sentencias del ejemplo de PASCAL se corresponden una por una con las del ejemplo de BASIC, si se exceptúa el hecho de que el PASCAL incluya un par de **begin** y **end**. Los **begin** y **end** son uno de los conceptos de *estructuración* del PASCAL. En programación, los conceptos favoritos vienen y pasan, pero el de la estructuración lleva ya varios años en la lista de los superventas. La razón de ello es fundamental: la única forma en que podemos esperar comprender un programa grande es construirlo a base de subestructuras menores y más simples.

Para hacernos una idea del objeto de **begin** y **end**, será instructivo considerar dos ejemplos de sentencias FOR en BASIC y en PASCAL:

BASIC	*PASCAL*
10 FOR K = 1 TO 10	**for** *k*:= 1 **to** 10 **do**
20 LET S = S + K	*s*:= *s* + *k*;
30 NEXT K	

BASIC	PASCAL
60 FOR K =1 TO 10	**for** *k*:= 1 **to** 10 **do**
70 LET S = S + K	**begin**
80 LET T = T + K * K	*s*:= *s* + *k*;
90 NEXT K	*t*:= *t* + *k***k*;
	end;

La sentencia FOR es la única construcción estructurada en el BASIC estándar mínimo. Cada FOR va acompañado forzosamente de un NEXT, y todas las sentencias comprendidas entre ambos se tratan como una sola unidad. En el PASCAL hay muchas construcciones estructuradas dentro de los programas, y **for** es una de ellas. Varias de estas construcciones utilizan las palabras **begin** (principio) y **end** (final) para encerrar a un grupo de sentencias que se ha de tratar como una sola unidad. A esta unidad se la conoce con el nombre de *sentencia compuesta.* La construcción **for** en PASCAL no constituye por sí sola una sentencia. Es una *cláusula* que ha de ir como prefijo de una sola sentencia. Si queremos que **for** afecte a varias sentencias, ponemos **begin** y **end** encerrándolas; esto las convierte en una sentencia compuesta, que cuenta como una sola sentencia. En el primero de nuestros ejemplos **for** precedentes, no necesitamos **begin** ni **end**, porque es sólo una sentencia la que se ha de iterar. Sin embargo, no importa si se ponen **begin** y **end** de sobra o redundantes, de manera que podríamos haberlos puesto. En nuestro segundo ejemplo, el **begin** y el **end** son necesarios, puesto que se ha de repetir más de una sentencia.

Todo el cuerpo de un programa PASCAL entero va encerrado por un **begin** y un **end**. El **end** final se escribe

end.

El punto significa que éste es el final mismo del programa. Al igual que ocurre con FOR y NEXT, **begin** y **end** se anidan en forma natural, por ejemplo:

```
begin (* A *)
   (* ... *)
begin (* B *)
   (* ... *)
end; (* B *)
   (* ... *)
begin (* C *)
   (* ... *)
end; (* C *)
   (* ... *)
end; (* A *)
```

Aquí, el **begin** que hemos marcado con la observación o el comentario (* *A* *) casa o se empareja con el **end** marcado similarmente, y así sucesivamente. Un buen programa se presenta gráficamente en tal forma que sea evi-

dente a la vista cuáles son los **begin** y **end** que van emparejados o casados, por ejemplo:

```
begin
   (* ... *)
   begin
      (* ... *)
   end;
   (* ... *)
   begin
      (* ... *)
   end;
   (* ... *)
end;
```

Desde luego, es un error que existan **begin** sin sus **end** correspondientes, y viceversa.

Te darás cuenta de que, dada la preponderancia que en el PASCAL tienen los conceptos de estructuración, el tener un poco de cuidado en la disposición o presentación gráfica del programa producirá grandes dividendos en legibilidad del mismo. Si tienes suerte, tu ordenador tendrá un programa o rutina de utilidad para lo que se conoce como *prettyprinting* (escritura bonita). Se trata de un programa que toma otro programa PASCAL y le da una disposición limpia y decente. (También en BASIC existen programas de este tipo, pero su tarea es menor y menos importante.) Sin embargo, aunque tales programas son útiles para lo que pudiera llamarse un "aseo final", es mucho mejor que te habitúes a dar a tus programas una disposición correcta ya desde el principio.

Sentencias individuales

Las sentencias individuales contenidas en nuestro programa de la *mediaaritmética* son similares en los dos lenguajes BASIC y PASCAL, por ejemplo:

BASIC	*PASCAL*
INPUT N	*read*(*n*);
LET S = S + X	*s*:= *s* + *x*;
PRINT "MEDIA="; S/N	*writeln*('*Media*=', *s/n*);

En PASCAL, desafortunadamente desde el punto de vista del programador de BASIC, se emplea *read* en la acepción de INPUT. En el PASCAL no

hay ningún concepto similar al de READ en BASIC, de manera que si, por error, empleas *read* para tratar de READ, pronto averiguarás que estás intentando lo imposible.

La sentencia de asignación del PASCAL no lleva la palabra LET, pero son muchos los BASIC que también permiten omitirla. Más dificultoso o, por lo menos, molesto, resulta el ":=" que hay que escribir en PASCAL en lugar del "=" del BASIC. (Esto afecta igualmente a las sentencias **for**, como puede verse echando un vistazo a los ejemplos precedentes.)

Las sentencias simples de salida del BASIC y el PASCAL son fundamentalmente similares. La palabra *writeln* es una abreviatura más bien desafortunada de lo que se podría haber llamado, pero no se llamó, *writeline*.

Finalmente, sin duda habrás observado que cada sentencia PASCAL lleva al final un punto y coma. En el BASIC, las sentencias se corresponden con las líneas del programa. En el PASCAL, el final de una línea se trata simplemente como un espacio, de manera que hace falta una marca explícita que indique el final de una sentencia. Si dispusiéramos de un terminal con una línea fantásticamente ancha, podríamos escribir todo un programa PASCAL completo en una sola línea.

A título de ejemplo más prosaico, las siguientes líneas de PASCAL

```
x:=
3;
y:= 4;
```

y

```
x:= 3; y:= 4;
```

tienen significados idénticos.

No obstante, normalmente es mejor no aprovechar la ventaja de la flexibilidad que el lenguaje ofrece. Hasta que te acostumbres al PASCAL y desarrolles un estilo propio en la disposición, lo más prudente es poner una sola sentencia en cada línea.

Para hablar con exactitud, el punto y coma más bien sirve para *separar* las sentencias que para terminarlas. Ello significa que no hace falta el punto y coma después de la última sentencia de un grupo, es decir, después de la sentencia que precede a un **end**. Si, a pesar de todo, ponemos punto y coma antes de **end,** como lo hemos hecho en todos los ejemplos hasta ahora, lo que hacemos en realidad es escribir una *sentencia nula* antes del **end**. Las sentencias nulas no producen acción alguna ni ocasionan ningún perjuicio. Si se te pega el dedo en la tecla del punto y coma y escribes

```
x=3;;;;
```

habrás escrito tres sentencias nulas, pero ello no afectará a tu programa. Al PASCAL, como a la mayoría de nosotros, no le importa no hacer nada.

Seguiremos ateniéndonos a nuestra convención de escribir punto y coma después de todas las sentencias (excepto después del último **end**, que lleva detrás un punto). La única excepción, como veremos, es que nunca ha de haber un punto y coma precediendo a un **else.** Sin duda, nuestro convenio molestará a los puristas del PASCAL, pero tiene el mérito de hacer más fácil la edición de los programas. Por ejemplo, es posible insertar una sentencia nueva antes de un **end** sin necesidad de añadir un punto y coma a la sentencia que anteriormente precedía al **end**.

Números de línea

El último punto es que los programas PASCAL no contienen números de línea. En el BASIC, estos números se usan para los GOTO y para editar. En el PASCAL, pocas veces se emplea el GOTO. Lo que importa más, y es una gran sorpresa para alguien que se haya formado en BASIC, el PASCAL no contiene ningún editor. Además, no lleva incorporados comandos u órdenes como RUN, SAVE, GET, OLD o NEW. En su lugar, todas estas posibilidades nos las da un entorno exterior llamado un *sistema operativo.* Los sistemas operativos son como los gobiernos. Nominalmente, existen para nuestro beneficio; pero, en la práctica, son grandes y poco ágiles, es difícil comunicarse con ellos y, en general, son un estorbo aparente para nuestro programa. Ya hablaremos de esto en el capítulo 3.

El PASCAL no es interactivo. En el BASIC, escribimos las líneas una a una en el compilador (cuando empleamos la palabra "compilador" en este libro, lo hacemos para designar lo mismo un compilador que un intérprete). En PASCAL, preparamos el programa completo usando un editor separado —esto se explicará más adelante, por si acaso no has encontrado la idea anteriormente— y luego se lo aplicamos a un compilador. Hay por ahí uno o dos compiladores PASCAL interactivos o semi-interactivos, pero hasta ahora no han tenido mucha difusión.

Sumario

Hasta ahora hemos encontrado tres conceptos en los que se pone mucho más énfasis en PASCAL que en BASIC. Son los siguientes:

- declaraciones,
- tipos de datos,
- estructuración.

Hemos hallado también que el PASCAL carece de las excelentes características que posee el BASIC para la edición, la grabación (SAVE), la carga (LOAD) y, en general, el manejo de los programas.

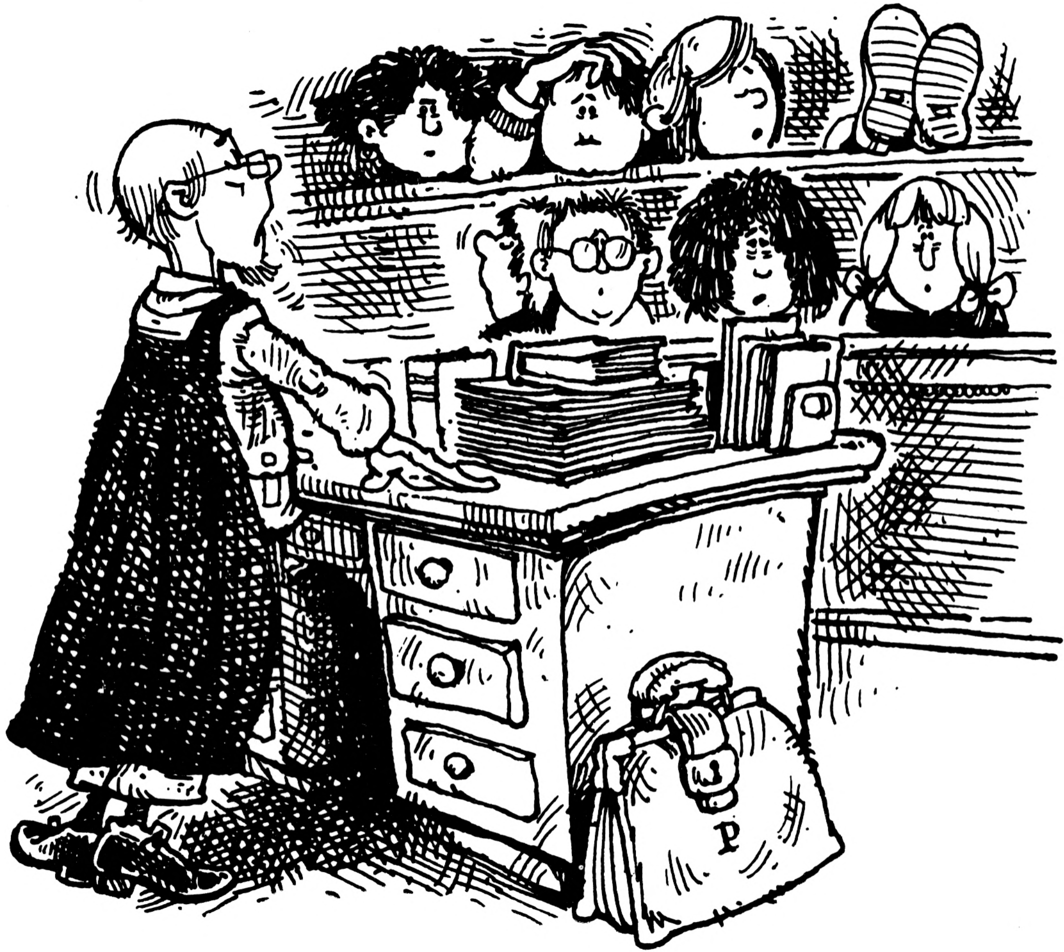
Algunos amigos

En este momento, hemos de presentarte a dos de nuestros amigos. Bill Mudd lleva veinte años programando en BASIC. Uno de sus programas tiene 10.000 líneas de BASIC, y funciona... por lo menos casi todas las veces. Sus comentarios sobre el material que hemos presentado hasta ahora en este libro no han sido muy estimulantes.

> ''A mí el PASCAL me parece un lenguaje para académicos con la cabeza perdida entre las nubes, que nunca han escrito un programa real en toda su vida'', nos dijo. ''Observe que el programa PASCAL dado como ejemplo es dos veces más largo que el correspondiente en BASIC. En especial, me divierten todas las cosas inútiles que hay que escribir. Hay que decir **program** al principio, por si acaso el ordenador piensa que lo que estás preparando es una lista para la compra. Luego hay que escribir otras varias líneas de morralla. Cuando finalmente se llega a la fase de la escritura del programa propiamente dicho, escribís **begin**, sólo para avisar al compilador que no habéis cambiado de idea, ni decidido dejar sin escribir el programa después de todo. Luego escribís prácticamente lo mismo que en BASIC, salvo que, como eso sería demasiado fácil, tenéis que añadir un montón de puntos y coma, dos puntos, paréntesis y cosas.''

Bastante alterados por esta inculta y grosera actitud de Bill, consultamos a nuestro otro amigo, el profesor Primple. El profesor es un fanático del PASCAL. Al igual que Bill, lleva veinte años escribiendo programas. Ha escrito soluciones para el problema de las ocho reinas en treinta lenguajes de programación diferentes, ha escrito funciones factoriales repetitivas en cuarenta y tres y, lo que es mejor, ha codificado la función de Ackermann en setenta y dos lenguajes.

> ''La conversión de los paganos siempre vale la pena, supongo'', nos dijo mientras se rascaba su barbita en gesto característico. ''Tendrá que trabajar mucho para convertir a los salvajes de Basiclandia a los modales realmente buenos del PASCAL. Es una pena que haya empezado por simplificar excesivamente muchos de los conceptos y que, peor todavía, haya animado a los salvajes a ensuciar la belleza del PASCAL con puntos y coma innecesarios y redundantes.''



No existe eso que llaman una traducción literal.

Un profesor de francés

2
Los objetivos del PASCAL

Ahora que hemos dado una idea de cómo son los programas PASCAL, vamos a estudiar en este capítulo los objetivos de este lenguaje y algunas de sus ventajas potenciales.

Historia

Uno de los hitos de la historia de la informática fue el desarrollo del lenguaje ALGOL 60. Los que, en nuestra época escolar, nos vimos torturados por unos profesores de historia que nos obligaban a recordar numerosas fechas, vemos con gusto el nombre del lenguaje ALGOL 60, puesto que el mismo nombre indica su fecha, 1960. El ALGOL 60 fue ideado y desarrollado por un comité de trece personas; pero, sorprendentemente, resultó una obra brillante que, desde entonces, ha influido siempre en el diseño de los lenguajes de programación. El ALGOL 60 propiamente dicho no ha conocido un uso muy extendido, tal vez porque se había adelantado a su tiempo, o quizá por-

que no había disponibles buenos compiladores en número suficiente; pero sus sucesores han conquistado el mundo. El ALGOL 60 contiene numerosas ideas buenas, pero una de las más importantes es la *estructura en bloques*. Más adelante estudiaremos este concepto. La idea es tan importante que, a causa de ella, el ALGOL 60 y sus sucesores han venido a ser conocidos como *lenguajes estructurados en bloques*.

El lenguaje PASCAL fue desarrollado a finales de la década de los sesenta por el profesor Nicklaus Wirth, de la Eidgenössische Technische Hochschule de Zurich. Es un lenguaje estructurado en bloques que continúa el camino iniciado por el ALGOL 60, introduciendo conceptos nuevos y simplificando otros ya existentes. Para 1973, el PASCAL había alcanzado ya su forma definitiva, y desde entonces ha conseguido un envidiable éxito. Al parecer, el profesor Wirth tenía en su pared un mapa del mundo en el que iba clavando alfileres sobre aquellos puntos en los que se utilizaba el PASCAL. Hoy, el mapa ha de estar tan perforado que seguramente habrá caído ya en pedazos. Además, este éxito del PASCAL no se debe a ninguna fuerte presión de ventas, ni al impulso de alguna organización poderosa. La gente ha elegido el PASCAL simplemente porque lo quería.

Sin embargo, el PASCAL no ha monopolizado el campo de los lenguajes estructurados en bloques. Entre otros lenguajes ampliamente utilizados figuran el C, el BCPL, el ADA y el PL/I. Una vez aprendido el PASCAL, hallarás que es relativamente fácil pasar a cualquier otro lenguaje estructurado en bloques, y viceversa.

Características específicas de diseño

Al llegar a este punto, vale la pena destacar dos características específicas de diseño del PASCAL que, dado que no las tienen sus competidores, han contribuido al éxito de este lenguaje. La primera es que el Pascal ha sido diseñado teniendo presentes los problemas de la realización o implementación. El resultado de ello es que los compiladores PASCAL funcionan en forma relativamente rápida, y producen un programa que también se ejecuta con rapidez. Además, los compiladores PASCAL no son tan gigantescos e incómodos como los de otros lenguajes estructurados en bloques, lo cual explica el gran uso que se hace del PASCAL en microordenadores.

La segunda de estas dos importantes características de diseño del PASCAL es que este lenguaje trata de ser *breve* y *flexible*. Una de las cualidades más grandes del diseñador de este lenguaje es que ha suprimido sin piedad todo rasgo potencial que no valiera la pena conservar. La recompensa obtenida ha sido un lenguaje fácil de aprender y de compilar. Sin embargo, no conviene ser despiadado a menos que las características principales del lenguaje tengan

la flexibilidad suficiente, que permita plegarlas y adaptarlas para llenar los huecos que deje la poda.

Se puede mantener (y nosotros mismos lo haremos) que algunas veces se han eliminado del PASCAL rasgos o características que no lo merecían, pero es difícil cuestionar el objetivo consistente en hacer un lenguaje pequeño o reducido.

Definición del PASCAL

En 1975, Springer-Verlag publicó el *Informe y Manual del Usuario de PASCAL*. Sus autores eran Kathleen Jensen y el propio Wirth. El "informe" es una definición bastante más formal del material presentado en el manual del usuario.

A lo largo de este libro, nos referiremos muchas veces al informe del PASCAL, será de gran valor para ti el hacerte con un ejemplar del libro de Jensen y Wirth; también constituirá un eficaz complemento del tratamiento un tanto informal que de dicho lenguaje se hace en este libro. Si eres una persona refinada, deberás buscar una de las primeras ediciones de Jensen y Wirth, con su bonita cubierta marrón y gris. Si tu personalidad se inclina más a un estilo de vida moderno y brillante, preferirás sin duda la edición que tiene las tapas plateadas y con salpicaduras escarlata.

El informe PASCAL sirvió como definición del lenguaje hasta que, en 1981, se produjo una norma internacional al respecto. La definición normalizada difiere muy poco del informe PASCAL, y en aquellos puntos en que hay diferencias, la mayoría de los compiladores actuales siguen a éste. En este libro nos referiremos alguna vez a la norma PASCAL, en aquellas situaciones en que ésta difiere del informe.

Las variaciones en el BASIC y en el PASCAL

Existen muchas realizaciones o implementaciones distintas del BASIC, que varían considerablemente entre sí en cuanto a los rasgos o características que ofrece cada una. Afortunadamente, sin embargo, la mayor parte de las versiones de BASIC poseen un núcleo central común, que es el que sirve de base para las explicaciones que damos en este libro. En realidad existe una norma ANSI para el BASIC mínimo, que en la actualidad va siendo cada vez más observada. (Se está realizando trabajo de normalización en algunos BASIC más

avanzados, pero, en el momento en que esto se escribe, su efecto en el ámbito de los usuarios ha sido muy escaso.)

También hay, desde luego, muchas versiones o realizaciones distintas del PASCAL, pero todas tienden a ser bastante similares, puesto que todas están basadas en el informe del PASCAL. Para cualquier compilador PASCAL que pienses utilizar, deberás consultar el correspondiente manual del usuario, al que denominaremos tu *manual local.* En tu manual local debe haber una sección que defina aquellos puntos en que tu compilador se aparte del informe o de la norma. Generalmente, hallarás una o dos ampliaciones y tal vez algunas restricciones esotéricas, pero nada de gran importancia. Como ya hemos dicho anteriormente, este libro no se ocupa de las ampliaciones no oficiales del PASCAL, por muy populares que puedan ser, de modo que para la descripción de ellas habrás de depender exclusivamente de tu manual local.

Cuando digamos en este libro que tal característica o rasgo ''no existe en el PASCAL'' o ''no existe en el BASIC'', nos referimos a las versiones estándar definitivas. Indudablemente, tú podrás descubrir versiones o realizaciones individuales de PASCAL o de BASIC que incluyan efectivamente tales características ''ausentes''.

Portabilidad

Si tu programa ha de lograr éxito y larga vida, ha de ser *portátil.* Esto quiere decir que el programa ha de poder moverse fácilmente de una máquina a otra, o entre dos compiladores diferentes en la misma máquina. Si evitas las ampliaciones no oficiales de los compiladores, hallarás que los programas PASCAL son bastante portátiles. Nadie que haya trasladado un programa grande de un ordenador a otro lo ha conseguido sin experimentar algunas dificultades o tropezar con algún problema. Sin embargo, tus problemas serán menos con el PASCAL que con la mayoría de los lenguajes.

En este aspecto, el BASIC es mucho peor que el PASCAL. Entre los distintos sistemas BASIC hay enormes variaciones, y habrá dificultades igualmente grandes cuando se trate de trasladar un programa BASIC largo de un sistema a otro. Tu única esperanza es limitarte al BASIC de un vendedor determinado, o atenerte exclusivamente al BASIC mínimo de la norma ANSI.

Método de ejecución

Si posees algunos conocimientos sobre *software* de sistemas, comprenderás que existe una diferencia entre un *intérprete* y un *compilador.* Si luego apren-

des mucho más sobre *software* de sistemas, hallarás que los dos conceptos no son tan diferentes como tú pensabas. En este libro no nos ocupamos de puntos tan sutiles. Por eso es por lo que describimos cualquier sistema BASIC o PASCAL como un "compilador", evitando así la constante repetición de "intérprete o compilador".

El tratamiento de un programa en el ordenador se hace en dos etapas o fases. Primero, el compilador convierte nuestro programa a una forma interna y, si existen errores de sintaxis, se nos advierten. Segundo, si la compilación se ha completado con éxito, se ejecuta (*run*) la forma interna del programa. Estas dos fases o etapas se denominan "tiempo de compilación" (*compile-time*) y "tiempo de ejecución" (*run-time*). (El programa del sistema que maneja nuestro programa durante la ejecución se llama el "sistema de ejecución", o *run-time system*.) Estas dos fases o etapas son completamente distintas en el PASCAL, aunque en algunos BASICs, especialmente los muy pequeños que no dan los errores de sintaxis hasta que se ejecuta el programa, la diferencia es menos evidente.

Algunas consideraciones prácticas

En un ordenador pequeño, tu elección de lenguaje puede estar determinada más por serias limitaciones prácticas que por puntos estéticos de orden más elevado. En tales términos prácticos, la comparación entre el PASCAL y el BASIC viene a dar los siguientes resultados:

1) *Tamaño de la memoria.* Un compilador PASCAL necesita mucha más memoria que uno de BASIC. A medida que pase el tiempo y la memoria de los ordenadores se vaya haciendo más barata, esta consideración irá perdiendo importancia.

2) *Velocidad de compilación.* La compilación en el PASCAL es más rápida que en la mayoría de los lenguajes estructurados en bloques, pero todavía tiende a ser más lenta que en los lenguajes estructuralmente más simples, como el BASIC.

3) *Velocidad de ejecución.* Inherentemente, la ejecución de los programas PASCAL no es ni más lenta ni más rápida que la de los de BASIC. Las diferencias que puedan hallarse procederán de la calidad de las versiones individuales.

Resumiendo, si tu única preocupación en la vida es el aprovechamiento avaro de la capacidad de memoria y del tiempo de máquina, es poco probable que el cambio desde el BASIC al PASCAL represente una ayuda para ti.

Puntos o cuestiones relativos a los lenguajes

Supongamos, sin embargo, que tus horizontes son más amplios que el de la avaricia extremada. Vamos a dedicar el resto de este capítulo a puntos algo más elevados y, en particular, a considerar las ventajas potenciales que el PASCAL tiene sobre el BASIC como lenguaje de programación.

El BASIC posee tres grandes ventajas. Es fácil de aprender, es fácil de usar y es bueno para la escritura de programas pequeños.

Cuando uno lleva algún tiempo programando, descubre que los programas grandes son fundamentalmente diferentes de los pequeños. Al hablar de ''grandes'', nos referimos a varios cientos de líneas. (Para un programador profesional, un programa ''grande'' podría ser de miles o incluso millones de líneas, pero nosotros seremos más modestos.) Cuando un programa se hace grande, hay una consideración que predomina sobre todas las demás: *el programa ha de ser fácil de leer y de entender, tanto para su autor como para los demás.* Si se consigue esto, casi todas las demás propiedades favorables se dan por añadidura. El programa se hace fácil de desarrollar, tanto inicialmente como cuando se hagan adiciones posteriores. El programa se hace fácil de depurar y de corregir. Se hace fácil de compartir: un grupo de personas puede colaborar en un programa, o tú puedes escribir un programa original tuyo y dejar que otros lo mejoren o amplíen, trabajando a partir de él. También se hace más fácil el trabajo de hacer que el programa funcione bien y en forma fiable, y el tener confianza en él. Finalmente, hay un argumento realmente invencible en favor de la legibilidad: si tu programa no es legible, nadie podrá darse cuenta de lo estupenda que es tu programación.

Si escribes un programa pequeño que no sea fácil de entender, ello no importa mucho. Todavía será posible seguirlo para ver qué es lo que trata de hacer. Si llevas las mismas costumbres de programación a un programa grande, naufragarás por completo, porque tu programa se hará imposible de entender para una mente humana ordinaria. Los programas de PASCAL son, inherentemente, más legibles que los de BASIC, y ésta es la primera razón que habrás de considerar al hacer el cambio. Tal vez en este momento no estés de acuerdo con esta aserción, pero lo estarás con toda seguridad cuando tengas una comprensión adecuada del PASCAL. Sólo como prueba, trata de hacerte con algunos programas en PASCAL (por ejemplo, de alguna revista o de alguna biblioteca) en un campo que te interese, y cuando tengas ya algo de idea del PASCAL como lenguaje, verás hasta qué punto son legibles los programas.

Mantenimiento

El argumento relativo a la legibilidad se puede reforzar si se considera lo que ocurre con un programa satisfactorio a partir del momento en que se ha escrito.

Probablemente no exista ningún programa que se haya utilizado durante años sin ninguna modificación. Por el contrario, todos los programas necesitan lo que se conoce como *mantenimiento.* En el mantenimiento entran la depuración o corrección de defectos y la introducción de pequeños cambios o adiciones. Tales cambios serán consecuencia de peticiones del usuario, o de cambios en el *hardware* o en el *software* con los que se ejecute el programa. Para un programa típico de producción, los costes de mantenimiento son muchas veces mayores que los de la escritura inicial del programa. Así, pues, el objetivo de un lenguaje de programación debe ser crear un programa que sea fácil de mantener; y, si es necesario, habrá de hacerse aun a costa de hacer más difícil su escritura inicial.

Si tú trabajas en un entorno de producción, tal vez haya un punto claro y definido en el que comienza la etapa de mantenimiento. Tú escribes tu programa y lo sometes a un proceso al que (risiblemente) se denomina "comprobación final". Cuando el programa ha superado esta prueba, pasa a otros para su utilización.

Al cabo de un tiempo muy corto, los usuarios vuelven con denuncias de errores o con peticiones de cambios, y, a partir de ese momento, empiezas a gustar las mieles y las delicias del mantenimiento.

Si, en cambio, escribes programas sólo para tu propio uso y tal vez para el de algunos amigos y colegas, la transición de la fase de desarrollo a la de mantenimiento es más gradual. Escribe una versión inicial, la usas un poco y luego decide en qué puntos deben hacerse modificaciones o desarrollos adicionales. Entonces produce una Versión 2. Llegará un momento en que te darás cuenta de que estás dedicando la mayor parte de tu tiempo a introducir cambios y pequeñas modificaciones, en lugar de a la escritura de rutinas completamente nuevas. Entonces habrás entrado en la fase de mantenimiento.

El mantenimiento puede ser realizado por la misma persona que escribió el programa, o bien por alguien distinto. En realidad, no hay mucha diferencia entre los dos casos. Todos hemos experimentado el fenómeno de mirar a uno de nuestros propios programas, escrito un año antes o cosa así, y pensar: "¿Quién será el idiota que escribió este programa? No entendiendo en absoluto qué es lo que trata de hacer." Es exactamente igual que si el programa lo hubiera escrito otro.

Incluso si pones en duda nuestra aseveración de que la legibilidad hace más fácil el desarrollo inicial de los programas, ya de entrada (y, desde luego, hay derecho a sospechar de la labia que derrochan en sus afirmaciones muchos libros), sin duda convendrás en que todas las ventajas de la legibilidad se ponen de manifiesto durante el mantenimiento de los programas.

Niveles de legibilidad

Existen varias formas diferentes de conseguir la legibilidad. Algunas de las importantes son las siguientes:

1) Hacer que cada sentencia sea fácil de leer.
2) Estructurar el programa.
3) Hacer que el programa esté estrechamente relacionado con el problema.
4) Imponer una disciplina en el estilo de programación.
5) Separar aquellos elementos del programa que puedan necesitar cambios.

Trataremos sucesivamente cada una de ellas, relacionándolas con las facilidades del PASCAL.

Hacer que las sentencias sean fáciles de leer

La ayuda más grande en que se puede pensar para hacer que las sentencias individuales sean comprensibles, es usar para las variables, subrutinas, etc., nombres que tengan el mayor significado posible. Consideremos las tres sentencias equivalentes que siguen en tres lenguajes, el BASIC, el PASCAL y el COBOL:

```
LET P3 = P1 + P2
pagatotal:= pagafija + pagahorasextra;
ADD OVERTIME_PAY TO FIXED_PAY GIVING TOTAL_PAY.
```

(Súmese la paga por horas extra a la paga fija, dando paga total.)

Evidentemente, la versión en BASIC es la más difícil de comprender. La versión en COBOL es interesante en cuanto que utiliza una sintaxis análoga a la inglesa. El lenguaje COBOL es muy utilizado por los programadores comerciales, y muy despreciado (un tanto injustamente) por todos los demás. Los programas son tan verborreicos que el equivalente del programa BASIC

```
PRINT 2 + 2
```

ocuparía alrededor de veinte líneas de COBOL. Una de las ideas originales en que se basó el COBOL era que los programas habrían de ser comprensibles para contables, gerentes, etc., que no saben absolutamente nada de programación.

Al hablar del COBOL nos hemos desviado para sugerir algunas ideas sobre la sintaxis de los programas. ¿Sirve de ayuda la notación en un lenguaje similar al inglés? Puede argumentarse que no sirve de ayuda en el mantenimiento de programas, porque es presumible que el encargado de este trabajo esté ya habituado a la notación de su lenguaje de programación. Lo que *sí que ayuda,* con toda seguridad, es el dar a los objetos manipulados unos nombres que indiquen su finalidad.

Estructuración

No es posible estar mucho tiempo en informática sin que le lancen a uno a la cara el concepto de "estructuración". Los vendedores de toda clase de productos de *software* proclaman que éstos son estructurados, y casi todos los artículos de *software* mencionan "programación estructurada". Sólo el profesor Primple ha escrito 158 comunicaciones sobre este tema; pero, desgraciadamente, no disponemos de espacio para reproducirlas aquí.

Indudablemente, toda la presión de ventas aplicada a la estructuración ha hecho que mucha gente reaccione negativamente ante este tema. Sin embargo, esa reacción no está justificada realmente. La estructuración es una *chica buena* de la programación. Le daremos el nombre de Sra. Buzz, por la abeja reina que, al imponer una estructura celular, consigue dirigir a miles de obreras para que construyan un conjunto coherente. La única forma en que podemos construir programas realmente grandes, y no digamos mantenerlos, es dividirlos en partes más sencillas, que puedan unirse entre sí para constituir el programa entero. Hay muchos métodos diferentes que caben bajo el encabezamiento de "programación estructurada". La elección entre las distintas alternativas tal vez no sea demasiado crucial, pero lo que sí es vital es que sigamos la directriz de la Sra. Buzz al tener alguna filosofía general, que sea práctica y coherente, para estructurar lo que hagamos. Sea cual sea la filosofía que escojamos, el PASCAL nos proporcionará herramientas que nos ayuden, aunque, indudablemente, no todo lo que querríamos.

Desde luego, hay muchos programas cuya escritura es obra de un equipo, y no de una sola persona. En estos casos, un lenguaje de programación no sólo ha de ayudar a distribuir el programa en subprogramas, sino que también ha de facilitar el que diferentes personas trabajen en los subprogramas sin interferirse. El PASCAL tiene un concepto de *ámbito local,* mediante el cual las variables pueden pertenecer a un subprograma y ser invisibles para otros.

Relación con el problema

Si programamos en BASIC, hemos de representar todo el mundo mediante variables que son del tipo de datos numérico o del de cadena (*string*). Sin embargo, hay muchos problemas del mundo real que no encajan naturalmente en tales tipos de datos. Si el programa simula el funcionamiento de un semáforo de tráfico, por ejemplo, el tipo de datos naturalmente asociado con él consiste en el juego de valores rojo, ámbar y verde (más rojo y ámbar juntos, en algunos países). Si representamos estos valores como 1, 2 y 3, nos perdemos algo. Esto es tan importante que nos lleva a presentar al primer *malo* de la

programación. Se trata del Sr. 869704, que nos estimula a pensar en números cuando los nombres serían mejor. Como veremos, el PASCAL tiene muchas formas de mantener a raya al Sr. 869704.

Imponer una disciplina

Uno de los placeres indiscutibles de la programación es la utilización inteligente de un lenguaje, o el empleo de trucos en él para hacer que el programa sea extracorto o extrarrápido. Aunque sea triste decirlo, estos placeres están más que contrarrestados por las agonías que producen cuando, durante la etapa de mantenimiento, hay necesidad de modificar un programa.

El PASCAL intenta imponer disciplina a los programadores al hacer que los trucos sean difíciles o imposibles y, en algunos casos, imponiendo una forma estandarizada de hacer las cosas. Los programas disciplinados son fáciles de entender.

Si has estado habituado a una mayor libertad, llegarás a considerar a tu compilador PASCAL como un maestro exigente y estricto. Ciertamente, uno tiene todavía la oportunidad de ser creativo, y hasta de divertirse, pero hay que seguir estrictamente las reglas del juego.

La disciplina tiene una finalidad, aparte de la de hacer los programas legibles. Está orientada a hacer que los errores sean menos. La correción de errores es una parte importante de los costes de mantenimiento de un programa, de modo que las economías potenciales en este aspecto son enormes. Si seguimos una disciplina de programación, nos será mucho más fácil comprobar si un programa hace lo que se pretende de él. Todavía sería mejor si hubiera métodos automáticos para verificar que los programas funcionan. El profesor Primple lleva muchos años trabajando en un sistema de este tipo. "Tengo un buen método, que funciona especialmente bien con el problema de las ocho reinas", dice, "pero, desgraciadamente, mis alumnos y mis colegas no tienen la inteligencia suficiente para comprenderlo."

Dado que tal vez no estemos capacitados para utilizar las herramientas de Primple, generalmente tenemos que hacer nuestra verificación mediante la laboriosa comprobación manual. Pues bien, la disciplina de programación es un regalo de los dioses, sea cual sea el método de verificación que se emplee.

Nos hemos encontrado a Bill Mudd sentado ante su terminal, y le hemos preguntado acerca de la programación ingeniosa e inteligente. "Me alegro de que me hayas preguntado eso," dijo con excitación. "Este programa de BASIC que tengo aquí lleva dos subrutinas que se solapan, la una desde la línea 1300 a la 1721, y la otra desde la 1529 a la 1961. ¿Ves?, una salta sobre el RETURN de la otra. Lo verdaderamente ingenioso es que las dos están semianidadas en un bucle FOR que hace que... No, espera un momento. Ahora que lo pienso, hace que..."

Separación de las partes que puedan necesitar cambios

Una faceta de un programa que muy bien puede necesitar la introducción de cambios la constituyen sus constantes. Aunque parezca paradójico, muchas veces las constantes no son constantes. Está claro que el valor de *pi* lleva mucho tiempo siendo 3,14159... e indudablemente continuará manteniendo su constancia. Similarmente, las probabilidades de que los catedráticos de Universidad reciban una paga adecuada seguirán siendo cero. Sin embargo, los programas frecuentemente contienen constantes que representan cosas tales como las dimensiones de una matriz o *array,* el ancho de una línea impresa, o el número máximo de registros que es posible procesar. Todas ellas es probable que cambien cuando se haga mantenimiento en el programa, o cuando éste se ejecute en un entorno nuevo y distinto.

El PASCAL tiene un medio para declarar las constantes, de modo que sea fácil cambiarlas más adelante. Esto tiene también la ventaja de que el significado de una constante arbitraria se vea claro también... a condición de que se dé a la constante un nombre sensato.

El cambio de constantes es, desde luego, una pequeña parte de la tarea total que supone el mantenimiento de un programa. Sin embargo, el PASCAL ayuda también a la Sra. Buzz al hacer que nos sea más fácil localizar los efectos de los cambios. Sin embargo, no seríamos leales si no te advirtiéramos que no puede uno pasarse en estos esfuerzos. Como principio general, si empleas una gran dosis de esfuerzo en la predicción de las partes de un programa que cambiarán y las que no van a hacerlo, tendrás tanto éxito como los economistas que tratan de predecir las finanzas de la nación.

Errores

Para la mayor parte de la gente, el trabajo de programación está dominado por una sola actividad: la depuración. Esto es aplicable tanto durante el mantenimiento como durante el desarrollo original del programa. El ocuparse de los errores o, mejor dicho, de la posibilidad de que existan o se produzcan errores, lo invade todo. Tanto la disciplina como la estructuración y la verificación contribuyen a reducir los errores, pero no los eliminan. (Hay una excepción cuando se trata de las actividades de programación del profesor Primple. Aparte de su herramienta automática de depuración, ha postulado una nueva disciplina de la programación estructurada. Está tratada en diecisiete de sus artículos técnicos más recientes y, según se dice, elimina los errores, especialmente en la función factorial.)

Aunque la mayoría de nosotros nunca puede esperar eliminar los errores, podemos hacer esfuerzos realistas para reducir los efectos de los mismos. El

principio más importante es que, cuanto antes se encuentre un error, más fácil será arreglarlo.

A continuación se da un espectro de los momentos en que pueden encontrarse errores:

1) antes de someter un programa al ordenador;
2) por el compilador;
3) inmediatamente que el error se manifieste durante una ejecución (*run*);
4) durante una ejecución, pero después que el error real haya producido una serie de efectos secundarios consiguientes;
5) al ver que las respuestas o soluciones son falsas;
6) por algún desastre subsiguiente en el mundo real, como resultado de respuestas o soluciones incorrectas del ordenador: nuestro avión, con una capacidad de diez litros en vez de diez millones de litros en el depósito de combustible, se ve implicado en un grave accidente.

La diferencia entre 3) y 4) se verá mejor por medio de un ejemplo. Si el error es un subíndice demasiado grande en una matriz o *array,* podrá ser detectado en el momento en que se emplee dicho subíndice. Sin embargo, algunos sistemas no detectan tal error, sino que en vez de ello dan al elemento de la matriz algún valor raro (*gash value*) y siguen a tontas y a locas. Entonces el error puede manifestarse algún tiempo después, cuando este valor extraño haya corrompido a otras muchas variables y el programa termine, por ejemplo, sacando la raíz cuadrada de un número negativo.

El objetivo de un lenguaje de programación, y el tuyo como programador, debe estribar en proveer una serie de redes de seguridad para cazar los errores tan pronto como sea posible. Será especialmente valioso si tu lenguaje de programación permite localizar en el momento de la compilación estos errores que potencialmente se pondrán de manifiesto al tiempo de la ejecución. El PASCAL facilita ayuda para esto.

Se empleará la expresión general *seguridad* para designar las técnicas encaminadas al descubrimiento temprano de los errores. El encargado de la seguridad es nuestro segundo *bueno.* Le llamaremos Perkins, como un hombre que conocemos y que es admirable. No esperes que Perkins evite todos los desastres; lo que sí consigue (y ya es algo de inmenso valor) es hacer que los desastres sean menos probables.

Evaluación

El examen precedente puede haber dejado la idea implícita de que, en las cinco facetas estudiadas, el PASCAL es lo más grande y el BASIC no vale para nada. Esta sería una opinión excesivamente simplista.

Para empezar, la mayoría de los BASIC contienen ampliaciones que proveen muchas de las facilidades mencionadas. De hecho, el éxito del PASCAL ha dado lugar a que muchas características del mismo se hayan infiltrado en el BASIC. Actualmente, casi todas las versiones de BASIC contienen bastantes más cosas que el BASIC mínimo de la norma ANSI. Tales ampliaciones están bien, pero adolecen de un problema que podremos apreciar mejor por medio de un ejemplo.

Uno puede empezar con una sencilla casa de un solo dormitorio y ampliarla gradualmente hasta terminar teniendo una catedral. Sin embargo, no es probable que el resultado final se parezca mucho a la catedral de Ely o a la de Chartres. De un modo análogo, las ampliaciones añadidas a los lenguajes de programación no llegan a formar un todo coherente y bien trabado; las probabilidades de éxito son mucho mayores si los objetivos de diseño se conocen desde el principio. (''Según mi guía'', dijo Bill, ''la Catedral de Ely se comenzó a construir en 1083, fue ampliada por diversos personajes sucesivamente hasta 1322, fecha en la que parte de ella se derrumbó, y luego fue reconstruida de una manera distinta.'')

En el transcurso de las últimas décadas se ha aprendido una importante verdad (la han aprendido aquellos que querían aprender). Dicha verdad es que también en el mundo de los lenguajes de programación tiene razón el refrán que dice que ''oficial de muchos oficios, maestro de ninguno''. Es mucho mejor tener distintos lenguajes de programación (idealmente, breves y flexibles) para cubrir diferentes aplicaciones. (Si estos diferentes lenguajes tienen un núcleo común, tanto mejor). Al igual que sucede con los seres vivos en la naturaleza, cada lenguaje de programación tiene su propio nicho ecológico en el que prospera. El BASIC lo hace allí donde los programas son pequeños, la capacidad de memoria es limitada, y los usuarios no tienen pretensión alguna de ser programadores. El PASCAL prospera cuando los programas son más grandes, las máquinas tienen mayores capacidades de memoria, y los usuarios, sin que sean necesariamente programadores de jornada completa, tengan preparación suficiente para poder emplear tiempo y esfuerzo en la programación. Como veremos, el PASCAL tiene también sus puntos flacos, y está superado por otros lenguajes en muchos entornos. Con muchos esfuerzos y molestias, se puede llegar a cultivar uvas en el Artico, o a criar pingüinos en el desierto; del mismo modo, es posible mantener artificialmente un lenguaje de programación en un entorno en el cual debería morir.

Los seres humanos y la disciplina

También es posible escribir buenos programas en BASIC y malos programas en PASCAL. Todos conocemos a gente de la que se puede garantizar que escribirá programas malos, inmantenibles y llenos de errores, sea cual sea el

lenguaje de programación que utilice. A la inversa, un buen programador generará un producto razonable incluso si tiene que usar el BASIC. Se pueden escribir programas disciplinados en BASIC, aunque hay que autoimponerse la disciplina. El excelente libro *The little book of BASIC style* (*El librito de estilo del BASIC*, Nevison, 1978) provee una base para esta disciplina.

Así pues, sólo se puede decir que el PASCAL ayuda a escribir programas disciplinados y fáciles de mantener, pero que los factores humanos tienen una importancia igual cuando menos a la de la selección del lenguaje de programación. Un buen libro sobre estilo y disciplina en el PASCAL es el titulado *PASCAL with style* (*PASCAL con estilo,* Ledgard, Hueras & Nagin, 1979).

El cambio en la forma de pensar

Llegamos ahora a la cuestión más importante que destacamos en este libro. *No sirve absolutamente de nada escribir un programa en PASCAL si se va a seguir pensando en BASIC.* En efecto, se pueden escribir programas en PASCAL que no son sino programas en BASIC con una sintaxis un poco diferente, pero eso no es en realidad usar el PASCAL. Lo que hay que hacer es cambiar la forma en que se piensa para resolver los problemas de la programación. Los mecanismos que ofrece el PASCAL, de los cuales aún no hemos mencionado la mayor parte, nos permiten enfocar los problemas y enfrentarnos a ellos en formas radicalmente nuevas. Por tanto, cuando hablamos de ''aprender PASCAL'', nos estamos refiriendo a algo mucho más importante que al simple aprendizaje de las reglas gramaticales y semánticas de este lenguaje.

Sabemos, desde luego, que no es fácil cambiar la forma de pensar. Como dijimos ya en el Prefacio, es *más difícil* aprender PASCAL sabiendo BASIC que sin saber programar absolutamente nada. El profesor Primple dice que el BASIC corrompe la mente. ''Lo malo del BASIC es que no tiene ningún valor'', declara riéndose con sorna, mientras los que le oyen gimen por dentro. (*N. del T.:* Hay en la frase anterior un juego de palabras intraducible en español, entre WIRTHLESS = "sin Wirth", creador del PASCAL, y WORTHLESS = ''sin valor o utilidad''; parece que es el juego de palabras precisamente lo que hace gemir a los oyentes.)

Finalmente, recordemos la cita que figuraba al comienzo de este capítulo. Lo que decía el profesor de francés era que si una traducción se hace literalmente, es decir, palabra por palabra y frase por frase, no será en realidad una traducción, pues no será correcta en absoluto. La traducción literal es nuestro segundo *malo* de la programación, con lo cual ya tenemos dos malos que se oponen a nuestros dos buenos. Este malo actúa insidiosamente sobre nuestros programas, llenándolos de tropiezos y cosas desagradables. Le llamaremos Frank Round (Redondo Franco), dándole el nombre producido por un programa de traducción automática para interpretar la expresión ''circuito abierto''.

Utilizar el (*aquí, el nombre de un conocido sistema operativo*), es como intentar mover una ballena muerta a patadas a lo largo de una playa.

S.C. JOHNSON

3
Sistemas operativos y editores

En el capítulo 1 dijimos que para usar el PASCAL necesitas entrar en un mundo que eludiste o rodeaste con el BASIC: el mundo de los sistemas operativos y los editores. Cuando este libro se haga especialmente aburrido u oscuro, te resultará agradable tomarte un descanso en su lectura y escribir algunos programas propios en PASCAL. A fin de que puedas hacerlo, vamos a tratar ahora de los sistemas operativos y de los editores. Si ya tienes conocimientos de estas cosas —como puede ser el caso de la mayoría de los lectores—, puedes saltar este capítulo y así pasar más rápidamente a saborear los placeres que te esperan más adelante.

Los microordenadores y los grandes ordenadores

Lo más probable es que hayas empleado el BASIC en un microordenador.

El PASCAL se inició como lenguaje en grandes y costosos ordenadores del tipo de unidad central, y sólo posteriormente llegó a estar disponible para los microordenadores. Todavía ahora, una importante proporción del uso del PAS-

CAL se hace en ordenadores grandes, aunque actualmente se va haciendo cada vez más difícil diferenciar entre uno de ellos y un microordenador.

No obstante, todavía existen diferencias en la filosofía del *software*. Un gran ordenador central suele dedicarse a atender a una enorme gama de usuarios distintos. Sus servicios y sus programas de utilidades están diseñados lo mismo para una compañía de seguros con veinte millones de pólizas que para usuarios individuales. El resultado es una inmensa mezcolanza que, como veremos, puede resultar difícil aprender. Un micro, en cambio, está desarrollado y equipado pensando mucho más en los pequeños usuarios individuales. El PASCAL puede ser similar en los dos tipos de máquinas, pero es más que probable que los medios y servicios y los programas de utilidades que le rodeen sean totalmente distintos.

Cuando uses el PASCAL por primera vez, puede que sea al mismo tiempo tu primera experiencia con el trabajo en grandes ordenadores centrales. Incluso si sigues usando el familiar microordenador, puede constituir una experiencia nueva, ya que habrás de adaptarte a utilizarlo en una forma con la que no estás familiarizado. El punto clave en el uso del PASCAL es que los programas se preparan en una forma que es completamente diferente de la del BASIC, y comenzaremos el capítulo explicando esta dicotomía.

Lenguajes interactivos y no interactivos

El BASIC es un lenguaje interactivo. El PASCAL es un lenguaje no interactivo. Al cambiar desde un lenguaje interactivo a uno que no lo es, tendrás la impresión de estar dando un paso atrás, porque la preparación y la compilación de los programas se convierten en un proceso mucho más aburrido y prolijo. Por tanto, si vas a enfrentarte a este cambio ha de ser con la confianza de que el PASCAL es un lenguaje mucho mejor que el BASIC. Una vez que se ha avanzado hasta el punto de ejecutar el programa, ya no existe mucha diferencia: puedes interaccionar con tu programa exactamente igual que con uno en BASIC.

La mejor forma de explicar la diferencia entre un lenguaje interactivo y otro que no lo es, es empezando con una analogía. Un lenguaje interactivo es como un supermercado, en donde hay toda clase de artículos o mercancías reunidos bajo un solo techo. No sólo puedes adquirir toda clase de productos u objetos, sino que puedes deambular al azar desde una sección del supermercado a otra y, suponiendo que no hayas pasado por la caja, puedes cambiar de idea en cuanto a las compras que ya hayas hecho. Por ejemplo, puedes estar en la sección de vinos y darte cuenta de repente de que no has elegido bien en la sección de frutas cuando cogiste la variedad de manzanas inadecuadamente denominada "Golden"; en tal caso, puedes volver a Frutas y Verduras

y enmendar el error tomando, por ejemplo, manzanas «reineta». Del mismo modo, si decides que no te puedes pasar sin comprar una barra de chocolate almendrado, aun cuando antes, con gran espíritu de renuncia, hayas pasado junto a ellas, sin tocarlas, puedes volver atrás y meterlas en tu cesta.

En el BASIC puedes preparar programas, listarlos, grabarlos, ejecutarlos, recuperar programas antiguos, etc. Todo ello se hace por medio de un conjunto de órdenes o comandos contenidos en el BASIC, tales como LIST, RUN, SAVE, etc. El uso de estas órdenes equivale a pasar a las distintas secciones del supermercado del BASIC. Además, en BASIC es fácil *editar* (es decir, modificar el programa, insertando nuevas líneas entre las existentes o sustituyendo algunas de éstas... exactamente igual que al cambiar las manzanas).

Un buen BASIC, en un ordenador con una capacidad de almacenamiento razonable, va en realidad más allá de nuestra analogía. Si has escrito inadvertidamente una sentencia incorrecta, tal como

LET A = B + /C,

se te avisa inmediatamente de tu error; no tendrás el problema de darte cuenta, después de salir del supermercado, de que has escogido mal los géneros comprados.

El empleo del lenguaje no interactivo se parece más a un grupo de pequeños comercios especializados, esparcidos por toda la ciudad. Uno es exclusivamente panadería y pastelería, otro carnicería y charcutería, otro frutería y verdulería, etc. Entras en la charcutería a comprar jamón para bocadillos. El jamón de la charcutería lo cortan para ti de junto al hueso y tiene un gusto estupendo, en contraste con el del supermercado, que se diferencia poco del plástico con el que lo envuelven. Sin embargo, si, al comprar el jamón, te das cuenta de que no compraste pan cuando estabas en la panadería, tendrás la molestia de tener que recorrer todo el camino de vuelta a ella para enmendar tu error. Además, suponiendo que te guste poner mantequilla en los bocadillos, tal vez tengas que volver también a la mantequería a comprarla.

El uso del PASCAL se parece a un recorrido por una serie de tiendas diferentes. Empezamos por la tienda en la que se escriben y se editan programas. Luego se pasa a la tienda del compilador para comprobar el programa y prepararlo para su ejecución. Si el programa está mal, hay que volver a la tienda de la edición; en caso contrario, se sigue a la tienda para la ejecución de programas. Si la ejecución produce resultados incorrectos, hay que ir todavía a otra tienda más: una que ayuda a localizar qué es lo que anda mal.

Preparación de un programa PASCAL

De un modo más específico, la sucesión exacta de operaciones para la preparación de un programa PASCAL es la siguiente:

1) Se escribe el programa PASCAL en un archivo. Ello se hace usando un editor, que es adecuado tanto para la preparación de programas nuevos como para modificar programas que existan ya en un archivo.
2) Tras dejar el editor, se introduce el archivo en un compilador. Este comprueba si el programa contiene errores en el uso del PASCAL. Si hay algún error, se vuelve al paso 1) y se usa el editor para corregir el programa. Si no hay errores, el compilador crea un nuevo *programa objeto,* que es una versión ejecutable del programa fuente original.
3) Se ejecuta el programa objeto.
4) Si el programa se para o interrumpe a causa de un error, tal como una división por cero o un subíndice ilegal, es posible que haya que entrar en un *debugger* o depurador que nos ayude a averiguar qué es lo que marcha mal.

Más adelante, en el curso de este capítulo, tendremos ocasión de estudiar en forma más detallada estas cuatro fases.

La idea importante que interesa retener es que el PASCAL sólo define lo que ocurre en los pasos o fases 2) y 3). El editor es un programa completamente independiente, que no está ligado en forma alguna al PASCAL; el editor puede emplearse indistintamente para escribir o editar programas en otros lenguajes de programación, o incluso textos en lenguaje corriente o datos numéricos. Igualmente, el depurador o ''debugger'' es un programa independiente, aunque en algunos sistemas se halla más ligado al PASCAL que el editor.

Sistemas operativos

El control fundamental del ordenador y de sus periféricos reside en un *sistema operativo.* Los dos componentes más importantes de un sistema operativo son su *sistema de archivo,* que se ocupa de nuestros archivos, y su *lenguaje de mando,* el cual (entre otras cosas) nos permite seleccionar cualquiera de las cuatro fases de la programación PASCAL antes mencionadas. Antes de estudiar en detalle estos dos componentes, empezaremos por enunciar unos cuantos hechos:

1) Los sistemas operativos y los editores varían mucho de unas máquinas a otras. Presentan al usuario unos interfaces que difieren radicalmente.
2) Algunos sistemas operativos son gigantescos, llegando a ocupar muchos millones de bytes.
3) En los últimos años, los sistemas operativos han ido haciéndose más fáciles de usar. Normalmente, los que se hallan en los microordenadores son recientes y, por necesidad pequeños; los de los grandes ordenadores centrales están muchas veces influenciados por rasgos o características que

quedaron anticuados ya en los años sesenta pero que, por razones de compatibilidad, todavía permanecen en el sistema. Como resultado de ello, los sistemas operativos de los microordenadores suelen ser mejores, aunque menos potentes, que los de sus gigantescos parientes.

4) Los sistemas operativos más grandes permiten que muchos usuarios "compartan el tiempo" (*time-share*) del ordenador, mientras que los sistemas más pequeños sólo pueden servir a un usuario en cada momento.

La tendencia actual es a ir abandonando el *time-sharing* o reparto del tiempo de máquina, y a ir utilizando cada vez más los ordenadores personales individuales. Existe, sin embargo, otra tendencia hacia el abandono del ordenador personal aislado y hacia la formación de redes de ordenadores personales conectados en conjuntos. Los ordenadores de la red trabajan o funcionan independientemente, pero pueden intercambiar archivos de información y compartir equipos o dispositivos caros, tales como impresoras y máquinas de composición tipográfica.

Lo ideal en un sistema operativo sería que éste hiciese que el entorno de la explotación del ordenador pasase inadvertido hasta el punto de serle invisible al usuario, el cual no debería tener conciencia de la existencia de otros usuarios empleando el ordenador en tiempo compartido, ni de si un dispositivo determinado se halla conectado a su ordenador directamente o por intermedio de una red... aunque, cuando se trate de recoger documentos producidos por una impresora, siempre viene bien saber dónde se encuentra situada ésta. Esperemos que los sistemas operativos de nuestros lectores se aproximen por lo menos a este ideal.

Sistemas de archivo

Para la mayoría de los usuarios, la tarea más importante que realiza un sistema operativo es la de cuidarse de sus archivos, los cuales pueden estar almacenados en uno o más discos o elementos análogos. A continuación se enumeran los servicios o facilidades más importantes de un sistema de archivo:

- *Nombres*. Si nuestro almacenamiento o memoria de reserva es capaz de mantener más de un archivo, hemos de disponer de un sistema para asignar un nombre a cada uno, de modo que siempre sea posible especificar el archivo que se desea en cada momento. Usualmente, un nombre de archivo está formado por un identificador tal como NOMINA o ENSEÑ1, seguido posiblemente por una misteriosa *prolongación* que dice al usuario el tipo del archivo de que se trate; por ejemplo, NOMINA.PAS puede ser un programa PASCAL, y ENSEÑ1.BAS un programa en BASIC. Asociado con los archivos va un índice o directorio, que da los nombres de todos los archivos existentes, por ejemplo, en un disco dado.

- *Ordenes.* El sistema operativo provee las órdenes o comandos que sean precisas para aquellas operaciones de archivo frecuentemente realizadas, tales como borrar, copiar o transferir a otro medio (por ejemplo, de disco a cinta).
- *Atributos.* Frecuentemente, los archivos llevan asociados determinados atributos. En una máquina con usuarios múltiples, puede haber algunos atributos asociados con la protección. El usuario debe tener la posibilidad de dotar a sus archivos de atributos que impidan que amigos indiscretos o entrometidos los modifiquen o tal vez incluso los lean o curioseen. Otros atributos pueden estar asociados con la permanencia de un archivo. Algunos sistemas proveen la posibilidad de archivos temporales que se anulan o borran automáticamente al final de cada sesión; esto contribuye a impedir que el sistema de archivo vaya almacenando una colección de cosas viejas e inútiles. Finalmente, puede haber atributos asociados con el *método de acceso.* Los archivos pueden ser *de acceso aleatorio* (*random-access*), lo que significa que uno puede leer o elegir partes de ellos en cualquier orden que desee; es algo semejante a un libro de referencia, como un diccionario o una guía telefónica, que normalmente no se leen de cabo a rabo, sino buscando determinadas voces o determinados nombres que se deseen consultar. Los archivos pueden, alternativamente, ser *secuenciales* (*serial*), lo que significa que para llegar a cualquier punto hemos de empezar desde el principio y explorarlos en forma ordenada y consecutiva (secuencial) hasta alcanzar el punto deseado. Hay, además de estos dos, otros muchos tipos de métodos de acceso, y ésta es una de las razones por las que algunos sistemas operativos ocupan millones de bytes; sin embargo, aquí no nos vamos a preocupar de ellos. En la práctica, la mayoría de los archivos se utilizan como archivos secuenciales y, en realidad, el PASCAL estándar sólo administra archivos secuenciales.

Lenguajes de mando (*command languages*)

En cierto sentido, el BASIC contiene un lenguaje de mando, que está formado por palabras tales como RUN, SAVE y LIST. Sin embargo, son tan simples que uno difícilmente pensaría que constituyen un lenguaje.

Los sistemas operativos proveen lenguajes de mando que generalizan las órdenes o comandos encontrados en el BASIC, y proporcionan asimismo otros servicios o facilidades. En realidad, el lenguaje de mando es el interfaz fundamental entre el usuario y los servicios o facilidades a su disposición. Muchos sistemas operativos soportan varios lenguajes de programación, y los comandos u órdenes están diseñados de modo que sean adecuados para todos los lengua-

jes. Por consiguiente, el PASCAL puede ser uno de tantos lenguajes, y las órdenes pueden no haber sido diseñadas pensando específicamente en el PASCAL.

Algunos lenguajes de mando son de una complicación y una verbosidad increíbles: acerca de algunos se han escrito libros enteros. Si ves que el ordenador que piensas usar tiene un lenguaje de mando descrito en un manual de usuario de 400 páginas, ejecuta el siguiente programa PASCAL:

```
open(ventana);
  repeat
    if bastantefuerte(usted) then
    begin
      coja(manual);
      arroje(manual, ventana);
    end;
    seleccionarnuevo(ordenador);
  until pequeño(manual);
```

Este programa, dicho sea de paso, introduce algunos conceptos de PASCAL que no hemos explicado todavía, pero esperamos que hayas cogido la idea básica de lo que se pretende.

Incluso cuando se trata de un lenguaje de mando sencillo, la mayor parte de los programadores se ven obligados a seguir aquella máxima defensiva que dice: no te aprendas el lenguaje entero, sino sólo las órdenes o comandos suficientes para ir arreglándotelas. (Posiblemente ya hayas hecho esto si has usado el BASIC con un sistema operativo, pero con el PASCAL será necesario que amplíes tu repertorio un poco más.) Ordenes o comandos típicos que podrías necesitar conocer son los siguientes; recuérdese, sin embargo, que los lenguajes de mando varían *inmensamente,* de modo que es posible que tu sistema operativo no se parezca en nada a esto.

EDIT *filename*	pasa al editor para trabajar en el archivo citado.
PASCAL *filename*	compila el programa PASCAL contenido en el archivo citado.
RUN	ejecuta el último programa compilado.
DEBUG	pasa al depurador para arrojar luz sobre lo que ocurrió durante la ejecución más reciente.

Niveles de comunicación

Un punto que da lugar a cierta confusión inicial entre los programadores de BASIC es que usualmente un sistema operativo ofrece diferentes niveles de comunicación. Cuando el BASIC nos pide una línea, podemos escribir un

comando o una línea de un programa. Cuando estamos usando un sistema operativo y un editor para preparar un programa PASCAL, puede haber tres niveles de comunicación separados:

1) Inicialmente estamos en el *estado de mando* (*command status*) en el sistema operativo. Aquí, la única cosa que podemos escribir es una de las órdenes del lenguaje de mando, para decirle al sistema lo que tiene que hacer a continuación. La orden o comando puede ser del tipo simple y autónomo, como la supresión o borrado de un archivo. En este caso, el sistema operativo obedece la orden y vuelve inmediatamente al estado de mando. Alternativamente, la orden dada puede ser una que pase el control a otro sistema, como la orden EDIT para pasar al editor.

2) Si pasamos al editor, entramos en el *estado de mando del editor*. Entonces hemos de escribir una de las órdenes o comandos que nos proporciona el editor, como puede ser una orden para insertar texto. Más adelante comentaremos las órdenes o comandos del editor.

3) Si escribimos una orden o comando para insertar texto, pasamos al *estado de entrada de datos*. Entonces escribimos el texto adecuado, en nuestro caso, un trozo de programa PASCAL. Al final del texto, se escribe un *carácter de control* que finalice la entrada de datos y nos devuelva al nivel 2).

Cuando hayamos terminado nuestro trabajo de edición, escribiremos un comando u orden de ''fin'', para volver al estado de mando del sistema operativo (es decir, al nivel 1)). A estas alturas, nuestro programa PASCAL deberá estar almacenado en un archivo. A continuación escribimos una orden que haga que nuestro programa se compile. Anteriormente hemos mencionado esta orden como PASCAL *filename*.

Los sistemas operativos y los editores son un ganado tan díscolo que todo lo que digamos acerca de ellos, por muy generales que sean los términos en que hablemos, tendrá numerosas excepciones. Existen, por ejemplo, algunos sistemas que no siguen la jerarquía de tres niveles que acabamos de mencionar. Hay editores en los que las órdenes o comandos se diferencian de los datos pulsando la tecla de control del teclado cuando queremos especificar que se trata de una orden; de otro modo, todo lo que escribamos será tratado como datos a insertar. Otros sistemas permiten el uso de comandos de nivel 1) en el nivel 2). A pesar de todo, nuestros tres niveles, aun cuando sean sólo conceptuales, pueden contribuir a clarificar tus ideas.

Puntos varios

La mayoría de los sistemas operativos modernos, y los lenguajes de mando asociados con ellos, ofrecen al usuario la comodidad de la *independencia de*

dispositivos. Esto significa que si una orden produce alguna salida, podemos, según nuestros deseos, enviar esa salida a un archivo, a nuestra terminal, o a una impresora. La realización de la orden o comando es independiente del dispositivo que usemos. Un mecanismo similar es aplicable a las órdenes o comandos que impliquen entradas.

Una orden para que se compile o ejecute un programa PASCAL puede implicar a muchos "archivos" de entrada y/o de salida. Un sistema dotado de independencia de dispositivos es una gran ventaja porque nos permite, cuando sea necesario, modificar la dirección en que se encaminan estos archivos.

Además del sistema de archivos y del lenguaje de mando, un sistema operativo puede tener otros servicios o facilidades que necesitas conocer. Tal vez tengas que *hacer tu presentación* al sistema al principio identificándote (nombre y clave o contraseña secreta), y *despedirte* al final. Pueden existir rutinas contables, que te facturen el importe de la utilización del ordenador. Puede haber también programas de utilidades o servicios, que ejecutan tareas tales como la limpieza de archivos, o la preparación de discos para su uso. Si tienes suerte, habrá una orden o comando HELP (ayuda), que te proporcionará asesoramiento cuando te metas en dificultades.

Finalmente, la cosa más importante que hay que saber antes de iniciar cualquier tipo de programación es la forma de parar si las cosas se te van de las manos, por ejemplo, si tu programa se mete en un ciclo infinito de salida. Todo sistema operativo tiene una tecla de interrupción o *ruptura* que se puede utilizar para abortar la actividad en curso. Normalmente, esta tecla ocasiona un retorno al estado de mando en el seno del sistema operativo. Antes de empezar a ejecutar tus programas, averigua la forma en que se puede hacer la ruptura.

Editores

Ahora que ya hemos descrito brevemente los sistemas operativos, haremos algo análogo con los editores.

El de la edición es un concepto sencillo. Incluye la creación de un archivo y el cambio o modificación de su contenido. Básicamente, sólo hay tres operaciones posibles: borrado o supresión, inserción y sustitución. El trabajo de edición es tan sencillo, en efecto, que cualquier programador de sistemas puede escribir un editor. La mayoría lo han hecho. El resultado es que existen miles de editores diferentes, ninguno de los cuales ha conseguido el predominio.

La gente es muy exigente y especial en el tema de los editores que usa. Es una característica del hombre (incluyendo, desde luego, a la mujer) que, cuanto más triviales son las cuestiones, más obstinadamente se aferra a su opinión. Así, hay gente que se emperra en usar un determinado editor porque emplea la S para indicar *sustitución,* y no otro que, en forma equivalente, use la C para indicar *cambio.* A causa de este prejuicio, algunos grandes ordenadores

que prestan servicio a cientos de usuarios tienen que mantener activos cientos de editores.

Si nunca has hecho uso de un editor antes de ahora, cuidado. Una vez que hayas aprendido tu primer editor, nunca volverás a ser la misma persona extremadamente razonable que eras. (El sistema empleado en el BASIC de usar los números de línea para especificar dónde ha de hacerse un cambio, por ejemplo escribiendo

```
100 LET X = 0
```

para sustituir a una línea 100 ya existente, es en cierto modo un editor, de manera que tal vez sea ya demasiado tarde para avisarte. Es más, algunos BASIC permiten desplazar un cursor sobre una pantalla para hacer modificaciones específicas dentro de una línea.)

Explicaremos algunas de las características generales de los editores. Los detalles (y, como hemos advertido, aquí es donde empiezan las dificultades) tendrás que consultarlos en el manual local de tu editor.

Rasgos o características de los editores

Lo primero que ha de hacer un editor es facilitarnos un medio para especificar en qué lugar del archivo deseamos introducir un cambio. Algunos editores antiguos exigían que los cambios se hicieran en el mismo orden en que se producían dentro del archivo (por ejemplo, si hubiéramos cambiado la línea décima, ya no podríamos hacer cambios en la novena), pero pocos de esos editores permanecen activos. Ahora, en cambio, el editor hace que el archivo a editar se parezca a uno de acceso aleatorio, aun cuando el archivo subyacente sea realmente del tipo secuencial. Existen tres formas básicas para especificar el lugar en que se han de hacer cambios:

1) *Por los números de línea.* Se trata del método menos atractivo. Uno dice algo así como "quiero cambiar la línea cincuenta y nueve". Si el archivo que se ha de editar es largo, éste es un método impracticable, a menos que tengamos un listado de aquél que lleve marcados los números de línea. Obsérvese que estos números de línea no son como los del BASIC, porque estos últimos están realmente en el texto del programa, mientras que los números de línea a los que nos referimos no lo están.

2) *Por el contexto.* Aquí tenemos el concepto de un *punto de exploración* que ha de desplazarse al lugar en el que haya de introducirse el cambio. Para hallar una línea determinada escribimos una cadena de caracteres que esperamos se produzca únicamente en aquella línea. El editor parte del punto actual de exploración y busca en forma secuencial hacia adelante (u,

optativamente, hacia atrás) hasta encontrar una línea que contenga la cadena de caracteres deseada. Entonces presenta visualmente esta línea, que se convierte así en el punto de exploración. Podríamos, por ejemplo, pedir al editor que halle una línea que contenga la cadena "$x+3$;". El editor se desplazaría desde el punto de exploración actual hasta que hallara dicha línea, y entonces la presentaría visualmente. La línea podría ser

cuenta: $= x + 3$;

Si ésta fuera la línea que quisiéramos, ahora estaríamos listos para introducir nuestro cambio. Sin embargo, si estuviera antes de la línea que deseábamos, podríamos continuar buscando la misma cadena, partiendo de este nuevo punto de exploración.

3) *Situando un cursor.* El más agradable de los métodos de edición consiste en señalar el lugar del programa en el que se desea introducir cambios. Posiblemente estés ya familiarizado con este método por haberlo usado en tu sistema BASIC. Para él es necesario disponer de un *display,* o presentación visual, con un cursor móvil. Lo más reciente en materia de edición consiste en disponer de un indicador o puntero electrónico —hay uno conocido como "ratón"— que se puede utilizar para desplazar el cursor sobre la superficie de la pantalla; es tan bueno que incluso los más llenos de prejuicios terminarán adoptándolo. La alternativa es mover el cursor sobre la pantalla pulsando en el teclado las teclas adecuadas.

Si tienes un programa de cierto tamaño (aunque sea pequeño), no entrará en una pantalla. Entonces puedes considerar que ésta es una *ventana* por la que se ve parte de tu archivo. Para desplazar la ventana a otra parte del archivo puede ser preciso usar la edición por contexto. Alternativamente, algunos editores realmente sofisticados permiten indicar en qué lugar del archivo ha de quedar la ventana mediante el desplazamiento del cursor a lo largo de una escala presentada visualmente en algún lugar de la pantalla.

Ordenes o comandos de edición

Una vez que hayamos encontrado el lugar del programa que se ha de modificar, estaremos preparados para escribir una orden de edición y realizar finalmente dicho cambio. Las órdenes más frecuentes son las fundamentales: inserción, supresión y sustitución. Estas órdenes pueden actuar sobre líneas completas de texto, o sobre subcadenas dentro de una línea.

A lo largo de los años, la tienda del editor (por volver a nuestra primera

analogía) ha ampliado la línea de productos que ofrece a sus clientes. Así, los editores modernos se pueden usar también para hacer operaciones tales como las siguientes:

- Sustituir por otra una cadena de caracteres cada vez que aparezca, a lo largo de todo un archivo. Por ejemplo, podríamos hacer uso del editor para sustituir el nombre de variable *scnt* por *shipcount*.
- Insertar algún texto almacenado en otro archivo.
- Trasladar texto de una parte a otra.
- Buscar una determinada combinación de caracteres todas las veces que se produzca.

Esta diversidad da a los editores muchas probabilidades de diferenciarse entre sí, y hay que decir que han aprovechado plenamente esta posibilidad.

Displays o sistemas de presentación visual perfeccionados

Cuando se está trabajando con información escrita sobre trozos de papel, suele ser conveniente tener ante la vista varias hojas al mismo tiempo. Al trabajar en un programa, por ejemplo, podemos tener el manual del usuario abierto en una página determinada, de modo que sea posible compararlo con un listado del programa y con la impresión de un mensaje de error que se haya producido al ejecutar dicho programa. Trabajaremos fijando la vista alternativamente en las tres hojas de papel.

En la actualidad muchas veces es difícil hacer lo mismo cuando se utiliza la pantalla de un ordenador en lugar de hojas de papel. Sin embargo, una de las tendencias más felices e importantes de los últimos años es la producción de sistemas de presentación visual en pantalla cada vez más potentes, los cuales, en conjunción con un *software* fácil de usar, nos permiten comunicarnos más fácilmente con los ordenadores. Ahora hay sistemas en los que es posible presentar en la pantalla varias ventanas separadas, y moverlas a un lado y a otro exactamente como si fuesen trozos de papel sobre una mesa de trabajo. Teitelman (1977) describe un magnífico sistema de este tipo.

Aunque tal vez resulte sorprendente, los sistemas de presentación visual que se encuentran en ordenadores personales baratos son a menudo más potentes que los conectados a grandes ordenadores que cuestan cientos de veces más. La razón de ello es que los costosos ordenadores de tiempo compartido son demasiado valiosos para dedicar su tiempo a ocuparse de una pantalla de presentación visual. Sin embargo, este fenómeno está desapareciendo ya, a medi-

da que se van haciendo más comunes las redes flexibles de ordenadores especializados.

La razón por la que nos hemos ocupado con cierta extensión de las presentaciones visuales es que, a medida que el trabajo de edición se va haciendo más sencillo, las diferencias entre la edición en PASCAL y la edición en BASIC pueden ir disminuyendo. El sistema utilizado por el BASIC, de usar los números de línea para editar, puede caer en desuso si resulta más fácil editar un programa señalando a una ventana sobre un *display*.

Utilización de un compilador de PASCAL

Una vez preparado o modificado nuestro programa mediante el uso de un editor, entramos en el compilador de PASCAL escribiendo un comando tal como

PASCAL *filename*

La mayor parte de los compiladores de PASCAL son completamente no-interactivos. Simplemente, uno les entrega un archivo y luego se sienta a esperar que salgan los mensajes de error. En el PASCAL sacarás muchos más errores de los que solías sacar en BASIC. Ello es así por tres razones:

1) El PASCAL es un lenguaje más elaborado.
2) En el PASCAL, un solo error puede dar origen a toda una multitud de mensajes. Si se te ha olvidado declarar una variable, puedes recibir un mensaje de error cada vez que se utilice dicha variable. Además, ciertos errores hacen que algunos compiladores de poca calidad se hagan un lío, de manera que dan varios errores falsos detrás de cada uno real. Así, algunas veces hallarás que, cuando corriges un error, los "errores" siguientes pueden desvanecerse en forma mágica.
3) Dado que los compiladores de PASCAL no son interactivos, es muy fácil cometer el mismo error muchas veces. En un nivel elemental, podrías olvidarte del punto y coma después de cualquier sentencia, y no sabrías nada de tu equivocación hasta que compilases todo el programa.

Por consiguiente, no desmayes cuando tu primer programa PASCAL, que ocupará tal vez diez líneas, genere veinte mensajes de error.

Un problema particular es el ocasionado por la facilidad que tiene el PASCAL para las construcciones que ocupan varias líneas, por ejemplo:

(* *un comentario*
de dos líneas *)

Si se te olvida el carácter "*)" que pone fin a un comentario u observación, el compilador escudriñará todo el programa para encontrar la "*)" siguiente, haciendo caso omiso de todo aquello sobre lo que pase. El resultado de ello puede ser una serie de desconcertantes mensajes de error. Problemas análogos pueden desprenderse de la omisión de comillas en *strings* o cadenas de caracteres, o de la de **end**. Por tanto, si hallas mensajes de error especialmente enigmáticos, explora estas posibilidades.

Cuando hayas llegado a la fase en que seas capaz de producir un programa correcto, el compilador PASCAL traducirá tu programa en un programa objeto. Este es el que ejecutas con RUN. Los programas objeto se pueden ejecutar tantas veces como se quiera. Una vez que tenemos un programa que funciona, no hay necesidad de compilarlo cada vez que se desee usarlo, ya que casi todos los sistemas PASCAL proporcionan la forma de conservar (SAVE) programas objeto en el sistema de archivo.

Ejecución de un programa PASCAL

Hemos supuesto que para ejecutar un programa PASCAL escribimos la orden

RUN

(RUN —o EXECUTE, como lo llaman algunos sistemas— podría ir seguido opcionalmente por el nombre de un programa objeto si quisieras ejecutar uno que no fuese el compilado más recientemente.) Algunos sistemas PASCAL inician automáticamente la ejecución si la compilación se ha completado en forma satisfactoria. Para ellos, la orden RUN es innecesaria.

La ejecución de un programa PASCAL es igual que la de uno en BASIC, hasta que las cosas van mal. Cuando se tiene un error en la ejecución (*runtime error*) o una ruptura, el PASCAL revela su naturaleza no-interactiva: normalmente, hace un listado (*dump*) de los nombres de todas las variables (excluyendo *arrays* o matrices y cosas análogas) y de los valores de aquéllas cuando se produjo el error. A continuación, abandona la ejecución. Entonces, uno utiliza el listado (*dump*) para tratar de descubrir qué es lo que ha pasado.

Depuradores o *debuggers*

Algunos sistemas operativos proveen sistemas depuradores interactivos, que se pueden usar en los programas PASCAL. Varios de éstos tienen prestacio-

nes similares a las encontradas en los buenos BASIC, por ejemplo declaraciones inmediatas.

Cualquier sistema depurador o *debugger* decente debe comunicarse con el usuario en términos de lenguaje fuente, es decir, en términos del programa que aquél escribió, y no en alguna representación interna que a él no debe interesarle. Sin embargo, todavía andan por ahí muchos sistemas depuradores que no son decentes, así que ya te puedes preparar a tener que aprender horribles detalles internos.

Algunas veces, un sistema depurador está integrado con la ejecución de programas, y proporciona facilidades tales como la traza o seguimiento (*tracing*), o la ejecución del programa sentencia por sentencia (una de cada vez).

Estas generalidades, un tanto vagas, son todo lo que podemos decir acerca de los sistemas depuradores. Su diversidad es tan grande que no serviría de nada entrar en más detalles.

Casos especiales

Como no hemos dejado de repetir a lo largo de este capítulo, uno de los placeres, y de las frustraciones, del *software* de ordenadores es que tanto sus filosofías básicas como los detalles de menor cuantía varían mucho. Los hay completamente heterodoxos y diferentes a los demás, y, a lo largo de los años, algunos de ellos han alcanzado tanto éxito que han llegado a convertirse en ciudadanos integrados en la comunidad del *software* y a gozar de gran predicamento en ella.

Aunque el BASIC es normalmente un lenguaje interactivo y el PASCAL no, ha habido excepciones. Los BASIC no interactivos son algo así como pájaros sin alas, y probablemente se extinguirán pronto. En cambio, los pocos intentos que se han hecho con los sistemas PASCAL semiinteractivos son más prometedores.

Hay además unos cuantos editores que están diseñados especialmente para ocuparse en forma exclusiva de programas PASCAL. Algunos de ellos están destinados a ayudar al usuario a preparar programas sintácticamente correctos. Véase, por ejemplo, el COPAS (Atkinson & North, 1981) o *The Cornell program synthesizer* (*El sintetizador de programas Cornell,* Teitelbaum & Reps, 1981).

A pesar de la importancia potencial de estos casos anómalos, no les prestaremos mucha atención en el resto de este libro, sino que seguiremos concentrándonos en el caso más sencillo y frecuente.

La opinión de un experto

Cuando fuimos a pedir a Bill Mudd su opinión acerca de los sistemas operativos, estaba todavía en su terminal trabajando en su ingenioso e inteligente programa BASIC. Acababa de sustituir

```
1096   GOSUB   4305
```

por

```
1096   GOSUB   4605
```

pero el programa seguía todavía sin funcionar. Tal vez esto explique su estado de ánimo, más bien agrio.

> ''En el BASIC, el cambiar tu programa y ejecutarlo de nuevo es una cosa trivial'', dijo. ''Sólo en este pequeño programa, he hecho yo unos cincuenta cambios y he repetido la ejecución otras tantas veces. Y ahora vienes a pedirme que pierda cinco minutos liado con sistemas operativos y editores para cada cambio.''
>
> ''Sin embargo'', continuó mientrás introducía un nuevo ajuste en su programa, ''me gusta realmente un comentario que has hecho: el consejo relativo a tirar por la ventana el manual si es demasiado grande. Y estoy observando que tu libro es ya bastante grande.''

Amigos míos, no volveremos ni remedaremos una antigua rabia, ni alargaremos la locura de nuestra juventud para que se convierta en la vergüenza de la vejez.

G.K. CHESTERTON

4
Traducción de conceptos del BASIC

Ahora ya hemos terminado con la información básica y preliminar, y vamos a dedicar el resto de nuestro libro a una descripción bastante completa del PASCAL.

No obstante, no queremos que el libro sea sólo un catálogo sistemático de rasgos y características del lenguaje. En lugar de ello, empezaremos con características que ya te son conocidas por el BASIC, y trabajaremos a partir de ellas. Los conceptos nuevos del PASCAL se introducirán a medida que surjan naturalmente en los ejemplos, más bien que en la posición impuesta tal vez por el informe PASCAL.

En este capítulo nos concentraremos en las sentencias individuales y en los constituyentes básicos que las forman, y en el capítulo próximo continuaremos con las subrutinas y las funciones. Ya hemos tratado en el capítulo 1 de las sentencias de asignación y de los rudimentos de la entrada/salida. (La sencilla sentencia de asignación empleada en el capítulo 1 es todo lo que ofrece el PASCAL en este aspecto. No hay sentencias de ''asignación mútiple'' como la

```
LET A = B = 0
```

que poseen algunos BASIC.)

Si deseas consultar la definición sintáctica precisa o exacta de cualquier sentencia PASCAL, mira el Apéndice C.

El espaciado y los comentarios u observaciones

Empezaremos, al nivel más bajo posible, con las reglas para la disposición de los programas PASCAL. Como ocurre en BASIC, se pueden añadir libremente los espacios que se quiera entre los símbolos de un programa para mejorar la disposición o presentación gráfica del mismo. En el PASCAL es obligatoria la inserción de un espacio para separar dos palabras o números adyacentes, por ejemplo en

for ↑ *count*: = *y* ↑ **to** ↑ 6 ↑ **do**

es necesario que haya por lo menos un espacio en cada uno de los cuatro lugares marcados por las flechas. No se permite la existencia de espacios dentro de los números o palabras.

Como se mencionó en el capítulo 1, el final de una línea se trata exactamente igual que un espacio.

También se trata como si fuera un espacio cada observación o comentario, que en el PASCAL se encierra entre llaves "{" y "}" como ya hemos visto. (Pueden usarse los símbolos "(*" y "*)" como alternativas para las llaves de apertura y cierre "{" y "}" si éstas no existen en nuestro teclado.) Los comentarios u observaciones pueden abarcar cualquier número de líneas, y producirse en el interior de sentencias, por ejemplo:

for *count*: = *y* **to** 6 (* *se para en* 6 *porque...* *) **do**

Nombres

Para nombrar un objeto, tal como una variable, se elige un *identificador*. Un identificador es una letra seguida de una sucesión de letras y/o cifras de longitud arbitraria. Normalmente, las letras minúsculas se tratan como diferentes de las mayúsculas. A continuación se dan algunos ejemplos de identificadores:

x, *x*1,*Southampton*6*Spurs*0

No eches en saco roto el consejo que te hemos dado: olvida el hábito del BASIC de usar nombres cortos; en lugar de ello, procura elegir nombres explícitos y con significado. Los identificadores formados por varias palabras se pueden hacer más legibles si el compilador permite el uso de mayúsculas y minúsculas, mediante el empleo de mayúsculas para las letras iniciales de las palabras. Por ejemplo, *NombreDeCuatroPalabras* es más legible que *nombredecuatropalabras*.

(En este libro, sin embargo, hacemos un uso bastante avaro de las mayúsculas en PASCAL, pero ello obedece únicamente a la convención que hemos adoptado de usar las mayúsculas para representar elementos o entidades de BASIC.)

Algunos compiladores de PASCAL sólo tienen en cuenta los ocho primeros caracteres de un identificador. Si en lugar de escribir *Southampton6Spurs0* hubiéramos tecleado *Southampton0Spurs6*, esos compiladores no habrían apreciado ninguna diferencia.

Los identificadores no han de contener ningún espacio. (''Yo creí que los nombres empleados en PASCAL debían tener un significado claro'', dijo Bill. ''Nuestraconversaciónnotendríaunsignificadomuyclarosinopudiéramosusarespacios.'')

Todas las palabras que forman parte de la sintaxis del PASCAL (tales como **var**, **for**, **to**, etc.) reciben el nombre de *palabras reservadas*. Estas son las palabras que, por razones de legibilidad, se imprimen convencionalmente en negritas cuando en un libro se incluyen programas de PASCAL. No se puede elegir como identificador una palabra reservada. La lista completa de las palabras que no han de usarse en esta forma es:

and	**donwnto**	**if**	**or**	**then**
array	**else**	**in**	**packed**	**to**
begin	**end**	**label**	**procedure**	**type**
case	**file**	**mod**	**program**	**until**
const	**for**	**nil**	**record**	**var**
div	**function**	**not**	**repeat**	**while**
do	**goto**	**of**	**set**	**with**

No te molestes en recordarlas todas; si alguna vez eliges por casualidad una de ellas para un identificador, el PASCAL te lo avisará muy pronto.

Tipos de datos

Antes de ponernos a considerar las sentencias individuales de BASIC, vamos a entretenernos en un área cubierta por el PASCAL, pero no por el BASIC: la de los *tipos de datos definidos por el usuario* y los *tipos de datos de subconjunto o subrango*. Nuestra finalidad al ocuparnos de esto en este momento es hacer que los ejemplos venideros resulten más ilustrativos.

El PASCAL nos permite definir nuestros propios tipos de datos. El hacerlo no es solamente una buena costumbre; es también algo divertido. Existe un goce creativo en hacer algo que, en la mayoría de los lenguajes, es imposible.

A fin de proveer algo de pasto o forraje para este capítulo y los siguientes, vamos ahora a definir algunos tipos de datos nuestros y completamente nuevos, para complementar los tipos, tales como *real* e *integer*, incorporados en el PASCAL. Estos tipos definidos por el usuario se declaran en la forma siguiente:

```
type
    howout = bowled, caught, stumped, runout, lbw;
```

La declaración de **type** precede inmediatamente a la de **var**.

Para aquellos lectores que desconozcan el juego del cricket, convendría explicar que el tipo de datos definidos por el usuario que acabamos de relacionar es una lista de las formas en que se puede eliminar o dejar fuera a un bateador. Podría utilizarse en un programa para la simulación de dicho juego. Desde luego, para la ejecución de tal programa sería necesario reducir en muchos órdenes de magnitud la velocidad del ordenador, a fin de acomodarla al *tempo* propio de este juego.

Cada tipo definido por el usuario es una sucesión de identificadores que representa a todas las constantes de dicho tipo. Así, las constantes del tipo *howout* (formelim) son *bowled* (derribado), *caught* (cogido)... *lbw* (piernaantelaportería). Estas constantes son analogías exactas de las constantes 1, 2, 3..., del tipo *entero* (*integer).* El objeto de un tipo definido por el usuario es hacer que el programa sea más legible. Una vez que se ha declarado un tipo, se pueden definir ya las variables correspondientes al mismo. Por ejemplo, se puede declarar a un bateador

```
var
    bateador: howout;
```

La variable *bateador* puede tomar cualquiera de los valores de nuestro conjunto; en realidad, éstos son los únicos valores que puede tomar. Así, podemos decir

```
bateador:= bowled;
```

Esto es muchísimo mejor que el estilo que el Sr. 869704 impone a los programadores de BASIC, consistente en representar *bowled, caught,* etc., por medio de números arbitrarios.

Obsérvese que los tipos definidos por el usuario que emplea el PASCAL son *completamente* diferentes de los enteros (*integers*), en la misma forma en que las cadenas o *strings* difieren de los enteros en BASIC. Por consiguiente, no se puede decir *bowled*+1, ni poner un *bateador* a cero.

(En la literatura del PASCAL es frecuente llamar a los tipos definidos por el usuario tipos *escalares* o *enumerados.*)

Tipos subrango

El PASCAL nos permite definir un tipo de datos que es un submargen o subrango de un tipo de datos existente. Por ejemplo, la declaración de tipo

```
enteropositivo = 1..maxint;
```

declara que *enteropositivo* es un nuevo tipo que constituye un subrango del tipo existente *entero* (*integer*). Este subrango empieza en el entero 1 y llega hasta *maxint,* el cual es una constante incorporada en el PASCAL, que representa el número entero más alto que pueda soportar nuestro compilador. Así, nuestro tipo subrango, como su nombre indica, está formado sólo por enteros positivos. Si a una variable de este tipo se le asignase un valor negativo o cero, ello sería un error.

Otros ejemplos de subrangos son:

(∗ *El tipo que sigue consta de los* 11 *valores posibles:* —5, —4,..., 4, 5, ∗)
enteropequeño = —5..5;
(∗ *El tipo que sigue excluye a lbw, que frecuentemente es objeto de discusion* ∗)
fuerasinduda = *bowled..runout*;

Los subrangos tienen tres usos principales. Primero, y más importante, se pueden usar para dar seguridad a un programa. Si Perkins sabe que en un determinado lugar se espera un entero positivo, cuando aparezca un entero negativo podrá eliminarlo y descartarlo inmediatamente como espúreo. En segundo lugar, los tipos de subrango pueden ser empleados por un compilador para ahorrar espacio de almacenamiento; un compilador podría, si le gustara la idea, almacenar un *enteropequeño* en un solo byte, o incluso en cuatro bits. En tercer lugar, los tipos subrango tienen un valor inapreciable para límites de *arrays* o matrices, como veremos.

Los tipos subrango son *compatibles* con los tipos que les incluyen. Ello significa, por ejemplo, que en todo lugar en que se pueda utilizar una variable *entero* o *integer,* se puede usar también una variable *enteropequeño* o *enteropositivo.* (Existe una pequeña excepción, de la que hablaremos en el capítulo próximo, cuando introduzcamos los "parámetros variables".)

Como ejemplo del trabajo de Perkins con los tipos de subrango, consideremos la sentencia

$p := q - 10;$

en la que p es un *enteropositivo* y q es un *enteropequeño.* El valor de q no puede ser superior a 5 y, por tanto, q —10 ha de ser negativo; así, el efecto de la sentencia sería tratar de igualar p a valor negativo, de modo que Perkins dé un mensaje de error. Algunos compiladores son suficientemente inteligentes para coger tales errores cuando se compila un programa, evitando así la molestia segura de comprobar al tiempo de la ejecución. Sin embargo, en la mayoría de los casos ha de hacerse la comprobación en la fase de ejecución; por ejemplo, en la sentencia

$p := q;$

ha de hacerse una comprobación en la fase de ejecución para asegurarse de que el valor asignado a p es positivo.

En cierto sentido, un tipo subrango es un tipo definido por el usuario; al fin y al cabo, el usuario lo define. Sin embargo, está basado en un tipo existente, y hemos reservado el término *tipo definido por el usuario* para aquellos que, como el *howout,* se definen desde cero.

Declaraciones

Todos los objetos del PASCAL, aparte de las constantes numéricas y las de cadena o *string,* se representan por medio de identificadores. En todos los casos, ha de haber una declaración de lo que significa el identificador. Así, en

```
type
    t = (ct1, ct2);
var
    v1: integer;
    v2: t;
```

se declara que el identificador *t* es un tipo; que *ct*1 y *ct*2 son constantes del tipo *t;* y que *v*1 y *v*2 son variables. (La primera es un entero o *integer,* y la última es del tipo *t.*) Con una excepción, que se explicará más adelante, todos los identificadores han de declararse antes de utilizarlos por primera vez.

Si se declaran varias variables del mismo tipo, se las puede combinar para no tener que repetir el tipo. Así

```
x1: integer;
x2: integer;
pig: integer;
```

se puede escribir

```
x1, x2, pig: integer;
```

Funciones incorporadas

El PASCAL, al igual que el BASIC, soporta un juego o conjunto de funciones incorporadas, y estas funciones tienen identificadores que las nombran (por ejemplo, *sin, cos*). A diferencia de las palabras reservadas, estos nombres pueden ser utilizados por el usuario para sus propios identificadores. Así, tu programa porno más vendido puede tener una variable que se llame *sin,* aunque entonces el programa no podría usar la función *sin* del PASCAL... pero, de todas maneras, no querría usarla. (*N. del T.*: Nuevamente tenemos aquí un juego de palabras intraducible entre "sin", seno trigonométrico, y "sin", pecado. Tal vez se

pudiera jugar en español con los dos significados de "seno", el trigonométrico y el anatómico femenino.) (Las reglas aplicables a los nombres de los tipos incorporados, tales como *real* e *integer,* y a las constantes incorporadas, tales como *maxint,* son similares.)

La mayoría de las funciones incorporadas en el BASIC tienen equivalentes en PASCAL, aunque hay varias cuyos nombres son diferentes. La lista que sigue da las traducciones:

BASIC	*PASCAL*
ABS	*abs*
ATN	*arctan*
COS	*cos*
EXP	*exp*
LOG	*ln*
RND	no existe
SGN	no existe
SIN	*sin*
SQR	*sqrt*
TAN	no existe

Todas las funciones PASCAL precedentes toman o aceptan un argumento real y entregan un resultado real, excepto la *abs,* que se puede usar con números enteros (*integers*) o con reales. La función de BASIC INT es similar a la función que en PASCAL se denomina curiosamente *trunc* (este nombre viene de "truncate", truncado); *trunc* sólo difiere de INT si el argumento es negativo: *trunc* (—3,2) es —3, mientras que INT (—3,2) es —4. La función *trunc* da siempre un resultado entero o *integer.*

Algunos PASCAL ampliados llenan las lagunas representadas por *no existe* en la tabla precedente.

La peor noticia para el programador de BASIC es que el PASCAL tiene una función *sqr* que significa el cuadrado de su argumento, y no su raíz cuadrada. El perversamente nombrado Sr. Sqr es un tipo realmente malo. Si construyes un puente basándote en ciertos cálculos de PASCAL, y el puente se cae, mira a ver si has dejado entrar en tu programa al Sr. Sqr, cuando en realidad querías meter *sqrt.*

Aunque resulte irónico, el *sqr* es el único atractivo adicional que el PASCAL ofrece entre sus funciones aritméticas estándar.

Constantes

Las constantes numéricas son idénticas en BASIC y en PASCAL, con la salvedad de que en este último siempre ha de haber por lo menos una difra delante de la coma decimal. Así, el ,1 del BASIC se convierte en el 0,1 del PASCAL.

Expresiones

En conjunto, las expresiones aritméticas son prácticamente iguales en PASCAL y en BASIC. Las pocas diferencias que hay son las existentes entre los tipos *integer* y *real,* y el uso del operador de potenciación.

En una expresión en PASCAL se pueden mezclar números enteros (*integers*) y reales, y obtener los resultados esperados. Así, si se suman un entero y un real, se obtiene un real. Lo que se ha de evitar principalmente es asignar un valor real a una variable entera o *integer*; en lugar de ello, se puede emplear la función *trunc* para convertir el real en entero antes de hacer la asignación. El operador de la división "/" siempre da un resultado real; si tenemos operandos enteros (*integer*) y queremos un resultado también entero, habremos de usar el operador **div**. Con ello se trunca el resultado, dejando sólo la parte entera. Así

7 **div** 4 es 1
5 **div** 4 es 1
5 / 4 es 1,25

El operador **mod** que, al igual que **div**, funciona con dos operandos *integer,* da el resto que queda cuando se divide el primer operando por el segundo. Así,

5 **mod** 4 es 1
7 **mod** 4 es 3

Si intentas escribir

x↑y

en PASCAL, la cosa no funcionará, porque el PASCAL no tiene operador de potenciación en absoluto. Lo que tienes que hacer es escribir

*exp(y*ln(x))*

Sin embargo, si se trata de una potencia entera, puedes hallar alguna forma más eficiente. Así, la expresión BASIC

X↑4

se puede escribir en PASCAL, usando la perversa función *sqr,*

sqr(sqr(x))

Como quiera que las funciones *exp* y *ln* son de ejecución lenta, mientras que la *sqr* es relativamente rápida, la expresión precedente se ejecuta mucho más de-

prisa que la forma más general. En realidad, la finalidad misma de la carencia de potenciación en el PASCAL es hacer que el usuario piense qué es lo que de verdad desea, y que entonces use el método más eficiente.

Sentencias IF

Ahora podemos pasar a considerar las sentencias BASIC individuales, y la primera de la que nos ocuparemos es la sentencia IF.

Si estás acostumbrado a escribir sentencias IF tales como

IF X = Y THEN 500

necesitarás cambiar tu forma de pensar cuando pases al PASCAL.

El problema no está en la *expresión de relación* existente entre IF y THEN (por ejemplo, en el caso anterior, X = Y), sino en lo que viene después de THEN.

Para empezar por lo más fácil, las expresiones de relación del BASIC son iguales en PASCAL, aunque es posible mejorar el programa utilizando los medios más avanzados que el PASCAL posee a este respecto. En realidad, el PASCAL permite una construcción más general, conocida como una *expresión de Boole,* o *Booleana* allí donde el BASIC permite una expresión de relación. Más adelante estudiaremos esta construcción..., por el momento, piensa en ella como en una expresión de relación.

Los *operadores de relación,* a saber:

=, < >, >, > =, <, < =

se escriben igual en BASIC y en PASCAL.

Y vamos con la gran diferencia. El PASCAL soporta dos formas de sentencia **if**. Se escriben así:

if <*expresión de Boole*> **then** <*sentencia*>

e

if <*expresión de Boole*> **then** <*sentencia*> **else** <*sentencia*>

La notación empleada es obvia. La sentencia que sigue a **then** se ejecuta si la expresión de Boole es verdadera (*true*), y la sentencia que sigue a **else,** si la hay, se ejecuta si la expresión de Boole es falsa (*false*).

No olvides que, en todas las situaciones del PASCAL en las que se necesite una sentencia, podemos meter varias sentencias encerradas entre un **begin** y un **end**. Hay una regla especial que ha de observarse en la sintaxis de las sentencias **if**: no se ponga *nunca* un "punto y coma" delante de un **else**. En otros casos, los "punto y coma" sobrantes, dentro de un orden, no importan. ("Ya le dije a usted que había cometido un inmenso error en el capítulo 1, cuando les recomendó que pusieran punto y coma al final de todas las sentencias", repetía el profesor Primple en un tono que indicaba que no toleraría ningún argumento en contra.)

A continuación damos, a título de ilustración o ejemplo, un breve fragmento de un programa en BASIC y su equivalente en PASCAL:

```
100  IF X > Y THEN 150
110  LET P = 0
120  GOTO 200
150  LET P = 1
160  PRINT "X ES MAYOR QUE Y"
200  ...
```

```
if x< =y then
   p: =0 (* ¡no pongas punto y coma aquí! *)
else
begin
   p: =1;
   writeln ('x es mayor que y');
end;
```

Vale la pena insistir aquí de nuevo en que a los programas PASCAL se les puede dar cualquier disposición que nos guste. Se puede argüir que la disposición que nosotros hemos empleado permite ver más claramente el significado, pero existen numerosas alternativas... incluyendo la de comprimirlo todo en una o en dos líneas.

Un segundo ejemplo, más sencillo, nos muestra un **if** sin **else**. En BASIC, es así:

```
100  IF P = 0 THEN 120
110  LET Q = 1
120  ...
```

mientras que su equivalente en PASCAL es

```
if p< >0 then
   q: =1;
```

En muchos BASIC se puede escribir algo muy similar a la precedente sentencia PASCAL. (En realidad, muchos BASIC ofrecen en su integridad el IF ... THEN... ELSE del PASCAL.)

La esencia del **if** en PASCAL consiste en pensar en forma positiva. En otras palabras, pensar en lo que quieres que se haga cuando la relación es verdadera (*true*); en el BASIC mínimo, por el contrario, frecuentemente se piensa en términos de lo que se salta, es decir, de lo que *no se hace,* si una relación es verdadera.

Saltos hacia atrás

En BASIC es frecuente que el número de línea que sigue a un THEN especifique un salto hacia atrás más bien que hacia adelante; estas sentencias IF no se pueden traducir al PASCAL en la forma arriba mostrada. Al parecer, necesitan un GOTO.

Esto nos lleva a un punto importante: los GOTO son ''malos'' de la programación. Y son tanto más insidiosos precisamente porque su aspecto es tan simple e inofensivo. Por lo menos, eso es lo que dice el profesor Primple. Los académicos revelaron la perversidad del GOTO en los años sesenta, y en nuestros días nunca se nombra a este ''malo'' en los centros de la ciencia y la erudición. En los años setenta hubo otros que siguieron la tendencia marcada por los académicos, y los duros hombres de negocios se convencieron de que el GOTO les estaba haciendo perder dinero, por lo cual se le debía despedir.

Cuando escribas en PASCAL, debes desterrar a los GOTO de tu mente. Un GOTO hacia atrás representa parte de un bucle, y se debe expresar utilizando las construcciones para bucles que presentaremos pronto. Un GOTO hacia adelante se puede expresar normalmente con sentencias **if**.

Si tomas un programa BASIC ya existente y lo traduces al PASCAL, verás que te resulta difícil eliminar los GOTO. Sólo podrás hacerlo si repites tus algoritmos en términos de construcciones de un nivel más elevado. Si has usado los GOTO durante años, la acomodación te será difícil, pero inténtalo de todos modos. Pronto te darás cuenta de que el pasarse sin GOTO es razonablemente indoloro. Tal vez una analogía apropiada sea el cambio desde escribir a máquina con un solo dedo, que tiene que andar saltando por todo el teclado, a mecanografiar correctamente con un método unificado, utilizando los diez útiles o facilidades que te proporcionan tus manos.

La ventaja que tiene la eliminación de los GOTO es que los programas se hacen más legibles. También se hace más facil el trabajo de verificar que los programas no contengan errores lógicos.

Sentencias GOTO

''Espera un momento'', dijo Bill, con una sonrisa triunfante. Sorprendentemente, sacó un libro de PASCAL que estaba oculto tras una

obra maestra titulada *Programas de guerras entre Galaxias.* "Veo que en PASCAL se pueden escribir GOTO. En realidad son igual que en BASIC, sólo que, como de costumbre, hay un montón de morralla redundante que especificar. Se puede escribir

goto 10;

y a continuación se puede escribir

10:

delante de la sentencia a la que se quiere ir con el GOTO. Se pueden utilizar los números de etiqueta que se quiera, sin que sea necesario que estén en ningún tipo de orden. La única molestia es que al principio mismo del programa —después de esa línea inútil de **program**— hay que escribir una línea de la forma

label 10, 20, 40, 67, ...;

relacionando todas las etiquetas que se hayan usado."

A decir verdad, el **goto** es útil de vez en cuando en el PASCAL, aun cuando el profesor Primple lo niegue. En realidad, los compiladores de PASCAL suelen estar escritos en PASCAL también, y algunos de ellos usan **goto**. El **goto** es especialmente útil en situaciones de error, en las que la acción natural es salirse de un salto del curso lógico actual y pasar a alguna acción completamente separada. Sin embargo, nosotros nos atenemos a nuestro consejo: lo mejor es tratar de desterrar los **goto** de nuestra forma de pensar, e introducirlos sólo como último recurso.

Sentencias FOR

Ya en el capítulo 1 vimos ejemplos sencillos del **for** en el PASCAL. En BASIC, la variable controlada, es decir, la variable cuyo nombre sigue al FOR, ha de ser del tipo de datos numérico. En el PASCAL, como veremos, puede ser de otros tipos de datos, pero *si es* numérico tiene dos severas restricciones. ("Querrás decir 'demasiado severas restricciones'," dijo Bill.) *(N. del T.*: Otro juego de palabras, entre las homófonas *two,* dos, y *too,* demasiado.)

La primera restricción, que ya mencionamos anteriormente, es que la variable controlada no ha de ser real. La razón de esta restricción es que la aritmética real es inexacta. Una sentencia de BASIC tal como

FOR K = .1 TO 1 STEP .1

es, por consiguiente, jugar con fuego. Si al sumar diez repeticiones de .1 se obtiene 1.0000001 en lugar de 1, entonces el bucle se ejecuta nueve veces, y no diez como se deseaba.

La segunda restricción procede de una de las facetas de la brevedad del PASCAL: no existe en él el equivalente del STEP del BASIC. El paso ha de ser 1 ó —1. En este último caso, se escribe **downto** en vez de **to** por ejemplo:

```
for k: = 1 to 10 do
for k: = 10 downto 1 do
for índice: = i + 1 to abs(q)do
for índice: = último downtoprimerodo
```

En los dos ejemplos últimos, al comienzo del bucle se toman los valores de *i, q, último* y *primero,* y ningún cambio que puedan experimentar estos valores dentro del bucle tendrá efecto alguno sobre el **for**. El BASIC tiene una regla similar. No se debe cambiar dentro del bucle el valor de la variable controlada, ni se debe suponer que después de que

```
for k: = 1 to 10 do (* ... *);
```

haya terminado, *k* va a tener necesariamente el valor 11.

No te desanimes por estas restricciones, ya que están sobradamente compensadas por otras dos sentencias de bucle que provee el PASCAL. La primera es la sentencia **while**. Tiene una sintaxis similar a la de una sentencia **if** sin **else**; la diferencia es que un **if** hace que una sentencia se ejecute una vez si una condición es verdadera, mientras que un **while** hace que una sentencia se repita continuamente mientras la condición siga siendo verdadera. El ejemplo que sigue lo ilustra.

```
cuenta: = 1;
(*  .
    .
    . *)
if cuenta < 11 then
begin
    writeln(cuenta);
    cuenta: = cuenta + 2;
end;
while cuenta < 11 do
begin
    writeln(cuenta);
    cuenta: = cuenta + 2;
end;
```

El **if** imprime el valor 1 e iguala *cuenta* a 3. El bucle **while** se ejecuta cuatro veces. La variable *cuenta* empieza en 3, como resultado del **if** precedente, y

se incrementa en 2 cada vez que se repite el bucle. Cuando *cuenta* llega a 11, la condición del **while** se hace falsa, y el bucle se interrumpe. Así los valores que se imprimen en el buche son 3, 5, 7 y 9. El **while** que acabamos de ver nos enseña también la forma en que se puede programar el equivalente de un **for** con un paso de 2.

La segunda construcción adicional de bucle es el **repeat.** El ejemplo que sigue muestra su sintaxis:

```
cuenta:= 3;
(*  .
    .
    . *)
repeat
    writeln(cuenta);
    cuenta:= cuenta + 2;
until cuenta> = 11;
```

El significado de lo anterior es obvio. El bucle **repeat** es semejante a un bucle **while**, con la salvedad de que la comprobación viene después del bucle, en vez de antes. En realidad, este ejemplo de **repeat** es otra forma de expresar nuestro anterior ejemplo de **while**. Obsérvese la sutileza de la sintaxis del PASCAL, que nos permite acortar el programa al no exigirnos que pongamos un **begin** y un **end** encerrando el cuerpo del bucle **repeat**, aun cuando el mismo implique más de una sentencia. Si piensas que esta regla te puede confundir, pon siempre el **begin** y el **end**, pues no tendrán ningún efecto perjudicial.

Todas las construcciones de bucle del PASCAL se pueden anidar, exactamente lo mismo que las sentencias FOR en BASIC.

Si tu forma de pensar está controlada por el BASIC, tu mente simulará un bucle con la adopción de una secuencia de valores para una variable, como en una sentencia FOR. Si haces el cambio al PASCAL, has de ensanchar tu mente y pensar ''bucle'' donde antes pensabas ''GOTO hacia atrás''. Tales bucles se escriben naturalmente usando **while** y **repeat**. Indudablemente, para empezar probarás primero con un **while** y luego lo cambiarás a un **repeat** o viceversa. Cuando hayas repetido este procedimiento durante algún tiempo, llegarás a saber instintivamente cuál de las dos construcciones utilizar en cada caso; la diferencia clave es que un bucle **repeat** se ejecuta siempre por lo menos una vez, mientras que con el **while** no siempre ocurre así. Para ponerlo de manifiesto, considera lo que ocurrirá en nuestros ejemplos anteriores si el contador (*count*) empieza con el valor 100, en vez del 3; el bucle **while** no imprime nada, pero el **repeat** imprime 100.

Para resumir, si estamos acostumbrados a trabajar con una sola herramienta y nos regalan un juego de tres para sustituirla, necesitaremos algún tiempo para adaptarnos. Si consideramos que los bucles constituyen frecuentemente la característica más importante de un programa, y que las tres herramientas del PASCAL son mucho mejores que la única del BASIC, veremos que vale la pena hacer un esfuerzo para adaptarse adecuadamente.

Sonidos y gráficos

Tras haber estudiado detenidamente la formación de bucles, llegamos ahora a un tema que se puede despachar con brevedad.

Hay muchos BASIC que disponen de medios para dibujar gráficos y producir sonidos. Tales medios faltan totalmente en el PASCAL, y apenas empiezan a introducirse en algunas de sus ampliaciones. Lo mismo ocurre con el PEEK y el POKE.

"Hay una regla muy sencilla acerca del PASCAL", dijo Bill. "Si algo es divertido, no se puede hacer."

"Me pregunto cómo será el superventas de los programas porno en PASCAL, sin ningún gráfico", añadió con una sonrisa afectada.

Las sentencias ON

La sentencia ON del BASIC

```
100  ON X GOTO 110, 150, 200
110  REM CASO EN QUE X = 1
120  LET Y = 3
130  GOTO 500

150  REM CASO EN QUE X = 2
160  PRINT Y
170  LET Y = 7
180  GOTO 500

200  REM CASO EN QUE X = 3
210  LET Y = 29

500  ...
```

se traduce en el PASCAl en una sentencia **case** de la forma

```
case x of
    1:
        y:= 3;
    2:
        begin
            writeln(y);
            y:= 7;
        end;
    3:
        y:= 29; (* hablando estrictamente, este punto y coma se debe omitir *)
end;
```

Los números 1, 2 y 3 *no son* etiquetas ordinarias. Bill no puede ir a ellas con GOTO. Se las conoce como *case labels* (etiquetas de la sentencia **case**), y son constantes del mismo tipo que *x,* es decir, en nuestro caso, *integer* o enteros.

La sentencia **case** es muchísimo mejor que la ON del BASIC. Pueden utilizarse como etiquetas de la sentencia **case** todos los valores posibles de *x,* y para un sólo **case** se puede poner una lista de etiquetas, separadas por comas, por ejemplo:

23, 19, 46:*y*:= 9; (* *etiquetas múltipes de* **case** *)

Las etiquetas de **case** pueden estar en cualquier orden; desde luego, no se puede emplear la misma etiqueta de **case** más de una vez dentro de la misma sentencia **case.**

Esto es una gran ayuda para la Sra. Buzz. El conjunto de la estructura, con los diferentes casos (*cases*) pulcramente separados, contribuye a mejorar la legibilidad de los programas.

El *case selector* o selector de casos (en nuestro ejemplo, *x*) puede ser cualquier expresión; puede producir un valor entero o, como veremos más adelante, valores de otros tipos, como carácter o *howout.* (En estas situaciones, las etiquetas de **case** no serían números. Para *howout,* por ejemplo, serían **stumped**, **bowled,** etc.) Esto contribuye en gran medida a la flexibilidad de las sentencias **case.**

El único problema en relación con las sentencas **case** es en el caso de producirse algún error. ¿Qué ocurriría en nuestro ejemplo si *x* tuviera el valor 4 y no hubiera ninguna etiqueta de **case** 4? Desafortunadamente, el PASCAL no define lo que ocurre en tal situación; un buen compilador producirá un mensaje de error, pero uno malo podría realizar cualquier acción imprevisible.

Sentencias BASIC sin equivalencias en PASCAL

Hay cierto número de sentencias BASIC que no tienen equivalencia en PASCAL.

La sentencia END no tiene equivalente, si se exceptúa tal vez el punto al final mismo de un programa PASCAL.

Un programa PASCAL se para automáticamente cuando llega al final. La mayoría de las realizaciones o implementaciones del PASCAL ofrecen una sentencia

halt;

que es equivalente al STOP del BASIC, pero no es completamente estándar.

De un modo análogo, ni la sentencia RANDOMIZE ni la función incorporada RND figuran en el informe del PASCAL, pero muchas implementaciones del lenguaje ofrecen facilidades similares.

Las sentencias DATA, READ y RESTORE no tienen equivalentes en PASCAL. Todas las entradas o introducciones de datos se han de hacer en una forma semejante al empleo de la sentencia INPUT del BASIC. DATA es muy útil en BASIC para inicializar tablas. El proceso para hacer lo mismo en PASCAL es mucho más pesado, hay que escribir sentencias explícitas de asignación, o bien introducir los valores desde un archivo.

La odiosa sentencia OPTIONBASE del BASIC está subyacente en el equivalente PASCAL de la DIM del BASIC.

Constantes, tipos y declaraciones

Finalmente, es útil presentar en este momento una facilidad del PASCAL que ya se anunció en el capítulo 2: la posibilidad de hacer fáciles los cambios en las constantes. Esto se hace en PASCAL dando un identificador a la constante como nombre, y haciéndole igual al valor de la constante. Todas estas constantes se agrupan en una sección **const** que viene al principio mismo de las declaraciones (''pero detrás de las etiquetas'', dijo Bill), de modo que se localicen fácilmente si es preciso modificarlas. A continuación aparecen algunos ejemplos de declaraciones de constantes:

```
const
    anchodelínea = 72;
    máximoderepeticiones = 20;
    carácterdeasentimiento = 's';
    (* las constantes pueden ser caracteres aislados *)
```

Luego, estos nombres se pueden usar en cualquier lugar del programa en el que se hubiera empleado la constante.

Del mismo modo que se dan nombres a las constantes, también se les pueden dar a los tipos. Puesto que es sensato dar un nombre a cada tipo, nosotros lo hemos hecho con todos los que introdujimos al principio de este capítulo, tales como *howout* o *enteropequeño*. (Más adelante presentaremos otros muchos tipos, tales como *arrays* o matrices y registros; también a todos éstos se les pueden asignar nombres.) Los nombres definidos por nosotros los usuarios se pueden utilizar en una forma semejante a la de los nombres incorporados en el PASCAL, como *integer* y *real*. En realidad, no hay necesidad de asignar nombres a los tipos, sino que se pueden escribir en forma explícita después de una declaración de variable. Así,

```
type
    dígito = 0..9;
var
    minúmero: dígito;
```

se puede escribir

```
var
    minúmero: 0..9;
```

No obstante, seguiremos ateniéndonos a nuestro consejo de asignar nombre a los tipos, aunque no comprendas la razón para ello hasta más adelante en este libro. (En algunas ocasiones, no seguiremos nuestro propio consejo; la excusa para ello será la de hacer los ejemplos autocontenidos o autónomos, y no dependientes de una declaración de **type** separada.)

Hasta ahora hemos mencionado cuatro posibles secciones de declaraciones. En el ejemplo que sigue se resumen (dando una declaración en cada sección) y se muestra la forma en que deben ordenarse.

```
label
    200;
const
    anchodelínea = 40;
type
    posicióndelínea = 1..anchodelínea;
var
    punterodelínea: posicióndelínea;
(* Ahora vienen las declaraciones de procedimientos y de funciones; véase
    el capítulo próximo *)
begin
(* Aquí empiezan las sentencias ejecutables *)
```

Cualquier sección que no se emplee, puede omitirse. Es fácil recordar el orden. Las etiquetas o *labels* vienen en primer lugar, de modo que Primple pueda ver inmediatamente si usas **goto**, y, si lo haces, ya no tendrá que mirar nada de lo que hayas escrito. En cuanto al resto, las **const** podrían ser necesarias para los **type**, y éstos serán necesarios con toda seguridad para las **var**, como muestra nuestro ejemplo; y esto fija el orden.

Facilitan la ida y el regreso.

Anuncio de billetes ferroviarios de ida y vuelta

5
Subrutinas y funciones

Conceptos básicos

Una de las mejores formas de dividir un programa grande en unidades manejables es por medio de subrutinas. La sencilla facilidad del GOSUB y el RETURN que proporciona el BASIC mínimo es tolerable para programas pequeños, pero se hace completamente inadecuada cuando se trata de programas grandes. (Esa es la razón por la que varios BASIC ofrecen facilidades más amplias.)

El PASCAL soporta tanto funciones como subrutinas; estas últimas reciben el nombre de *procedures,* o procedimientos. Las funciones del PASCAL tienen algún parecido con las del BASIC, especialmente con las funciones multilínea que ofrecen algunos BASIC. Un procedimiento PASCAL es muy parecido a una función PASCAL; puede pensarse en él como en una función que no produce ningún resultado. Por consiguiente, se parece mucho más a una función multilínea de BASIC que al mecanismo del GOSUB y el RETURN.

Empezaremos dando un ejemplo de una función multilínea BASIC que halla el máximo común divisor de dos enteros, X e Y. Esta función se ha tomado, con autorización de John Wiley and Sons, del excelente libro sobre BASIC escrito por Kemeny & Kurtz (1980).

```
110  DEF FNE (X,Y)
120      REM ESTA FUNCION HACE USO DE UN ALGORITMO
           DEBIDO A EUCLIDES
130      LET Q = INT(X/Y)
140      LET R = X—Q*Y
150      IF R = 0 THEN 190
160          LET X = Y
170          LET Y = R
180          GO TO 130
190      LET FNE = Y
200  FNEND
```

La función PASCAL equivalente es como sigue:

```
function MáximoComúnDivisor(número1: enteropositivo;
                            número2: enteropositivo): enteropositivo;
(* Esta función produce el máximo común divisor del número1 y el
   número2. El método usado es el algoritmo de Euclides. *)
var
   resto: integer;
begin
   repeat
      resto:= número1 mod número2;
      if resto< >0 then
      begin
         número1:= número2;
         número2:= resto;
      end
   until resto= 0;
   MáximoComúnDivisor:= número2; (* resultado de la función *)
end; (* MáximoComúnDivisor *)
```

Lo que precede es una declaración de la función, y se coloca con las demás declaraciones. Específicamente, las declaraciones de funciones y de procedimientos se colocan después de las declaraciones de variables, es decir, después de la sección que lleva el encabezamiento **var**.

La llamada a una función PASCAL se hace en una forma similar a la de una función BASIC, por ejemplo:

```
x:= y + MáximoComúnDivisor(y + 6,q);
```

Argumentos y parámetros

Si escribimos en BASIC

```
DEF FNX(P) = ...
 .
 .
 .
PRINT FNX(L + 9), FNX(Q9)
```

a la variable P usada en la definición de FNX se le llama el *parámetro,* y a las expresiones tales como L + 9 y Q9 que se usan en las llamadas de FNX se les llama *argumentos.* El parámetro es una especie de elemento ficticio. Cada vez que se llama a la función, el valor del parámetro toma el valor del argumento correspondiente.

El PASCAL funciona en una forma muy similar, con la salvedad de que la terminología que se utiliza es diferente: a los argumentos se les llama *parámetros reales,* y a los parámetros se les denomina *parámetros formales.* Desde luego, cada uno es libre para llamar a las cosas como guste, y nos parece que será mejor, en este libro, conservar la sencilla terminología del BASIC.

Aplicando esta terminología al *MáximoComúnDivisor,* los identificadores *número*1 y *número*2 son parámetros; y, en nuestro ejemplo de llamada, $y + 6$ y q son los argumentos correspondientes a *número*1 y *número*2, respectivamente.

Sólo por mantener la imparcialidad, haremos al PASCAL el favor de respetar su convención, en virtud de la cual se llama a una subrutina un procedimiento.

En el PASCAL, a diferencia del BASIC, cada procedimiento o función puede tener tantos parámetros como uno quiera. Así, una función podría no tener ningún parámetro, y otra podría tener cinco. Si esto es así, nunca tendremos que suministrar ningún argumento cuando llamemos a la primera función, y siempre tendremos que suministrar cinco cuando llamemos a la segunda. Al hacer la llamada de la función, se asigna al parámetro el argumento, razón por la cual ambos han de ser compatibles en cuanto a tipo. Las reglas son exactamente iguales que las de una sentencia normal de asignación. Por tanto, un argumento entero (*integer*) puede servir para un parámetro real, pero no a la inversa. A las funciones incorporadas les son aplicables unas reglas idénticas; así, por ejemplo, aunque *sqrt* pide un argumento real, de hecho aceptará uno entero y lo convertirá a forma real entre bastidores.

Puntos a observar

Son muchos los puntos a observar en relación con la función *MáximoComúnDivisor.* El primero es que la función parece enteramente un programa PASCAL completo (si exceptuamos el uso de ";" en vez del "." después del **end** final). Esta característica de diseño es deliberada y muy afortunada, ya que las funciones y los procedimientos constituyen un medio para dividir o descomponer un programa en subprogramas más pequeños. Es conveniente considerar a un programa como un procedimiento; una ejecución (*run*) del programa es equivalente a una llamada a este procedimiento. El programa tiene como parámetros los archivos que se usan para comunicar con el mundo exterior. Esa es la razón por la que la primera línea de un programa es

program *nombreprograma*(*input, output*);

Los nombres *input* y *output* son nombres internos del PASCAL para los archivos de entrada y salida por omisión (*default*). Estudiaremos más esta cuestión en el capítulo 9. La parte ejecutable de la función va encerrada entre un **begin** y un **end**. Esto se parece a una sentencia compuesta, pero seguiremos necesitando el **begin** y el **end** aun cuando la función esté formada por una sola sentencia.

El segundo punto es que las declaraciones son locales para la función. Esta es una idea realmente importante, y más adelante en este capítulo le dedicamos mucha atención.

El tercer punto se refiere a la finalidad de la línea de encabezamiento

function *MáximoComúnDivisor*(*número*1: *enteropositivo*;
*número*2: *enteropositivo*): *enteropositivo*;

Este encabezamiento, que hemos repartido en dos líneas porque es bastante largo, da primero el nombre de la función. No tenéis obligación de usar nombres tan largos como *MáximoComúnDivisor,* pero los nombres largos, hasta cierto punto, ayudan a la legibilidad. El nombre de la función va seguido por las declaraciones de sus parámetros, encerradas entre paréntesis. Estas declaraciones se escriben exactamente en la misma forma que la lista de declaraciones que se escribe en la sección **var**; obsérvese que no se pone punto y coma después de la última declaración. Nuestros dos parámetros, *número*1 y *número*2, son ambos del tipo *enteropositivo*, el tipo de subrango que definimos en el capítulo anterior. En realidad, podíamos haber acortado estas declaraciones dejándolas en

(*número*1, *número*2: *enteropositivo*)

El elemento final de encabezamiento da el tipo de datos del resultado de la función. (Por una caprichosa casualidad, en este ejemplo el *enteropositivo*

se produce tres veces seguidas; en general, como veremos, pueden ser tres tipos diferentes.) Por tanto, la forma general del encabezamiento de una función es

function <*nombre*>(<*lista de parámetros*>): <*tipo*>;

En las funciones que no tienen ningún parámetro, se omite la lista de parámetros indicada entre paréntesis.

Todos los tipos de datos usados en las declaraciones de funciones han de ser nombres de tipos. No se puede, por ejemplo, escribir un 1..10 explícito. Esta era una de las razones por las que anteriormente te aconsejamos que asignaras un nombre a todos los tipos que definas.

La razón de que empleemos como tipo de datos de los dos parámetros *enteropositivo* mejor que *integer,* es para ayudar a Perkins. Si se llama a la función con un argumento negativo o cero, Perkins puede señalar inmediatamente la equivocación cometida. (En realidad, si hubiéramos permitido usar cualquier entero [*integer*] como argumento, la función sólo fallaría cuando el segundo argumento fuera cero... y ello, porque **mod** intentaría dividir por cero. La función daría un resultado si cualquiera de los dos argumentos fuese negativo, aunque es dudoso que este resultado fuese el deseado por el usuario. Es como si un cliente pidiese en un restaurante rosbif y natillas; el resultado puede conseguirse, pero un camarero atento no aprobaría la elección de menú.)

Las ventajas de usar también *enteropositivo,* en vez de *integer,* como tipo de datos del resultado de la función, no son tan patentes. Sin embargo, ello proporciona alguna información adicional, tanto a Perkins como a cualquier lector del programa. Y cuanto más sepa Perkins, más seguros estamos.

Los dos ejemplos de declaraciones de funciones que siguen a continuación, deberán arrojar más luz sobre los encabezamientos de funciones.

```
type
   poso0integer = 0..maxint; (* un entero positivo o cero entero *)
var
   leecuenta: integer; (* cuenta de los números leídos por la función
                            leecubo *)
(*    .
      .
      .   *)
function potencia(X:real;K: poso0integer): real;
   (* Esta función produce la k-ésima potencia de x *)
var
   resultado: real;
   índice: integer;
begin
   resultado: = 1;
   for índice: = 1 to k do
      resultado: = resultado * x;
   potencia: = resultado;
```

```
end; (* potencia *)

function leecubo: real;
   (* Esta función lee un número real y lo eleva al cubo. También
      cuenta el número de números leídos *)
var
   númeroleído: real;
begin
   read(númeroleído);
   leecubo:= potencia(númeroleído,3);
   leecuenta:= leecuenta + 1; (* cuenta los números leídos *)
end; (* leecubo *)
```

La función *potencia* muestra parámetros cuyos tipos son diferentes.

La función *leecubo* muestra una función sin ningún parámetro. Nos muestra también que una función puede llamar a otras funciones y puede hacer referencia a una variable declarada fuera de sí misma (*leecuenta* en el ejemplo precedente). Todos estos puntos tienen paralelos exactos en BASIC.

Estas dos funciones ilustran también un buen hábito de programación: poner un comentario al comienzo de una función para explicar lo que ésta hace, y otro después del final de la función indicando el nombre de la misma.

Cómo trabaja la función

Si ahora examinamos la forma en que trabaja nuestra función original *MáximoComúnDivisor,* veremos que utiliza un **repeat** en lugar del GOTO hacia atrás del BASIC. El programa resultante de ello es más fácil de comprender. Sin embargo, es un poco menos eficiente en su ejecución, ya que la condición "*resto* = 0" se verifica o comprueba dos veces. Este es un fenómeno común; algunos bucles tienen la forma

Comienzo del bucle
   Acción 1
   Verificación de la condición terminadora y, si es verdadera (true), *salida del bucle*
   Acción 2
Fin del bucle

De estos bucles puede decirse que se ejecutan "N veces y media", para algunos N. Algunas veces se les puede convertir cómodamente en bucles ordinarios variando la redacción del algoritmo; de no ser así, sólo se pueden expresar en PASCAL introduciendo pequeñas ineficiencias como la precedente, u, osemos decirlo, mediante el uso de un **goto**. Las ineficiencias son un precio que

la mayor parte de la gente está dispuesta a pagar. Algunos lenguajes tienen una sentencia especial que dice "salida del bucle actual"; esto resuelve el problema.

Finalmente, observamos que en nuestra versión PASCAL no hace falta la función BASIC INT, ya que de todos modos estamos trabajando con enteros o *integers*. Puede argumentarse incluso que el trabajar con números reales, como lo hace el BASIC, es una forma antinatural de resolver este problema.

Toda llamada a una función *f* ha de ejecutar como mínimo una sentencia de asignación de la forma

```
f: = < expresión > ;
```

Obsérvese que el uso del nombre de una función, *f,* en el término izquierdo de una asignación es bastante especial; cualquier otro uso de *f* supondría una llamada a *f.* La última asignación de este tipo a *f* define el resultado que se ha de entregar. Tal es la finalidad de la última línea de *MáximoComúnDivisor,* que es

```
MáximoComúnDivisor: = número2;
```

En PASCAL no existe la sentencia RETURN. La función o el procedimiento vuelven automáticamente al programa principal tan pronto como llegan a su **end**, del mismo modo que el programa principal se termina automáticamente cuando llega a su propio **end**.

Procedimientos

Un procedimiento se declara exactamente en la misma forma que una función, con la salvedad de que la línea de encabezamiento se escribe así:

```
procedure < nombre > ( < lista de parámetros > );
```

Al igual que ocurre en las funciones, la lista de parámetros entre paréntesis se omite si no existe ningún parámetro. A continuación tenemos un procedimiento para imprimir los números 1 a *n,* cada uno en una línea:

```
procedure papelmojado(n: integer)
(* imprime los números 1 a n *)
var
   índice: integer;
begin
   for índice: = 1 to n do
      writeln(índice);
end; (* papelmojado *)
```

La llamada a un procedimiento se escribe como una llamada de función, con la salvedad de que una llamada de procedimiento es una sentencia por sí misma, por ejemplo:

> *papelmojado*(6);
> **if** *x* > 0 **then**
>   *papelmojado*(*q* + 3);

Dado que los procedimientos son similares a las funciones, gran parte de lo que digamos acerca de unos será también de aplicación a los otros. Por consiguiente, durante el resto de este capítulo emplearemos el término genérico *rutina* para hablar de un procedimiento o una función, indistintamente. Es un término tomado del lenguaje BCPL.

Procedimientos incorporados

En el capítulo precedente presentamos algunas de las funciones incorporadas en el PASCAL. Además, el PASCAL tiene cierto número de procedimientos incorporados. Muchos de ellos tienen relación con las entradas/salidas, y, efectivamente, el **writeln** que ya hemos utilizado en varios ejemplos es uno de ellos. A medida que progresemos en la lectura de este libro, encontraremos otros varios procedimientos (y funciones) incorporados. En el Apéndice A se da un sumario de todos ellos.

Algunas de las rutinas incorporadas del PASCAL ilustran una flagrante injusticia. Los programadores de sistemas que definen las rutinas incorporadas disponen de medios o facilidades que a nosotros, los usuarios comunes, nos están prohibidos. En particular, las rutinas incorporadas tienen argumentos que se omiten opcionalmente, y, como ponen de manifiesto los procedimientos de entrada/salida, pueden tener unas listas arbitrariamente largas de argumentos de cualquier tipo. Podemos explotar estas facilidades cuando llamamos a rutinas incorporadas, pero no podemos usarlas cuando definimos nuestras propias rutinas.

Declaraciones locales

Una rutina puede tener declaraciones que sean locales, o exclusivas de la misma. Entre el encabezamiento de la rutina y su **begin**, podemos definir un conjunto o juego de declaraciones de **label**, **const**, **type**, **var** y, desde luego, declaraciones de rutinas anidadas. En la función *MáximoComúnDivisor* he-

mos sido modestos en cuanto a nuestras declaraciones locales. Existe simplemente una sola variable, llamada *resto*. Además, los parámetros *número*1 y *número*2 están tratados como declaraciones locales, de modo que tenemos un total de tres variables locales. Estas declaraciones locales son una de las características más importantes del PASCAL. Significan que las rutinas pueden ser autónomas, o autocontenidas. Ello tiene tres grandes ventajas:

1) No hay necesidad de elegir nombres singulares y únicos para las variables locales usadas en una rutina. Si cualquiera de los nombres que usemos está utilizado ya fuera de la rutina, nuestra utilización local del mismo tiene precedencia sobre su utilización exterior. (En el BASIC hay un atisbo de esto cuando se emplea una declaración como

   DEF FNA(X) = 3 — X/Y

   Aquí el nombre X es local para la declaración de la función FNA.) El uso local puede ser completamente diferente del uso externo; el primero, por ejemplo, podría ser un **type** y el último un **var.**

2) La Sra. Buzz puede poner a cada obrera a escribir una rutina diferente, sin que unas interfieran a las otras.

3) Las rutinas son más fáciles de depurar y de mantener, porque se reducen los efectos del exterior. Un principio muy sano en el diseño de programas es la idea de la *reserva u ocultación de información,* debida a Parnas (1972). El aforismo de Parnas es que los detalles de la forma en que representamos y manipulamos cada juego o conjunto de objetos deben estar confinados a unas pocas rutinas; esta información detallada queda oculta para el resto del programa, el cual trata a estas rutinas como "cajas negras".

Refinamiento progresivo

Hemos dicho que las rutinas pueden contener en sí mismas las declaraciones de sus propias rutinas locales. Esto contribuye a la estructuración de los programas y, en particular, al método de estructuración conocido como *refinamiento progresivo* (*stepwise refinement*), que fue introducido por el mismo Wirth (1971). En este método se proponen primero unas rutinas muy potentes, y se resuelve el problema utilizándolas. A continuación se toma cada una de estas potentes rutinas y se la codifica utilizando rutinas más pequeñas. Este proceso se prosigue hasta llegar a rutinas sencillas que se puedan codificar directamente. Un ideal adoptado por muchos programadores es que el programa principal y el cuerpo de cada rutina no excedan de una página de texto, digamos unas cincuenta líneas. El resultado de ello es que el programa, por grande que sea, será relativamente fácil de leer. No te preocupes, por tanto, si defines una rutina a la que luego se llama una sola vez; si esta rutina hace que tu programa sea más legible, se ha ganado el derecho a la permanencia.

El ámbito local

En PASCAL, todas las declaraciones tienen su *ámbito*. El ámbito de una declaración es la rutina en la cual se produce, junto con todas las rutinas que sean locales de la primera, y cualesquiera otras que estén anidadas aún más profundamente en el interior de estas rutinas locales, y así sucesivamente. (Las declaraciones hechas en el programa principal tienen como ámbito propio el de todo el programa, llamándose "de ámbito global".) Fuera de su propio ámbito, una declaración es desconocida, y es posible usar el mismo nombre para significar algo completamente diferente. En esto existen dos casos posibles: puede reutilizarse un nombre en ámbitos disjuntos y no solapados, o bien se puede reutilizar en un ámbito incluido en el de una declaración anterior. En este último caso, como hemos dicho, la declaración más interna toma precedencia sobre la(s) más externa(s), dejándola(s) en suspenso. Estos principios se ilustran en el ejemplo que sigue. Los comentarios incluidos en el programa explican las reglas relativas a los ámbitos; para hacer más fácil la lectura, los nombres se han encerrado en un cerco en el sitio en que se declaran.

```
program usa(input, output);
(* Los nombres declarados en el programa principal son todos de esta-
   dos de los EE.UU. *)
var
   vermont: integer;
   ohio: real;
(* < < < < < < < < < < < < < < < < < < < < < < < < < < < < *)
procedure oregon(lincoln: integer; eugene: real);
(* Las variables locales son nombres de pueblos y ciudades de Oregon *)
const
   bend = 36;
type
   newport = (astoria, albany);
var
   salem: newport;
   foster: 1..bend; (* un pueblo muy pequeño *)
(* < < < < < < < < < < < < < < < < < < < < < < < < < < < < *)
procedure portland(fremont: newport);
(* Las variables locales son nombres de calles de Portland *)
var
   lombard: integer;
   union: real;
   vermont: 1..123; (* no tiene relación alguna con el vermont anterior *)
   foster: 1..bend; (* no tiene relación alguna con el foster anterior *)
```

```
begin
(* Aquí todas las declaraciones anteriores están dentro del ámbito, ex-
   cepto las de vermont y foster externas, que han quedado en suspenso
   al tomar precedencia las locales del mismo nombre *)
   (*  .
       .
       .  *)
end; (* portland *)
(* Esto pone fin al ámbito de fremont, lombard, union, y los vermont
   y foster internos *)
(* Los vermont y foster anteriores ya no están en suspenso, y entran de
   nuevo en el ámbito *)
(* > > > > > > > > > > > > > > > > > > > > > > > > > > > *)
begin (* cuerpo de oregon *)
   (*  .
       .
       .  *)
end; (* oregon *)
(* Esto pone fin al ámbito de lincoln, eugene, bend, astoria, albany,
   newport, salem, foster, portland *)
(* > > > > > > > > > > > > > > > > > > > > > > > > > > > > *)
(* < < < < < < < < < < < < < < < < < < < < < < < < < < < < *)
procedure maine (lincoln : real ; bangor: integer);
(* Las variables locales son nombres de pueblos y ciudades de Maine *)
(* El lincoln arriba escrito no tiene relación ninguna con el anterior *)
var
   portland: 1..6; (* no tiene ninguna relación con el portland anterior *)
   augusta: real;
begin
   (* Aquí las declaraciones que se encuentran dentro del ámbito son ver-
      mont, ohio, oregon, maine, lincoln, bangor, portland, augusta *)
   (*  .
       .
       .  *)
end; (* maine *)
(* Esto pone fin al ámbito de lincoln, bangor, portland, augusta *)
(* > > > > > > > > > > > > > > > > > > > > > > > > > > > > *)
begin (* cuerpo del programa principal *)
   (* Aquí las declaraciones dentro del ámbito son vermont, ohio,
      oregon, maine *)
   (*  .
       .
       .  *)
end.
```

Obsérvese especialmente el efecto que las reglas relativas al ámbito tienen sobre los nombres de las rutinas. Sólo se puede llamar a una rutina cuando su nombre esté dentro del ámbito. El nombre está dentro del ámbito durante toda la rutina o todo el programa en los cuales haya sido declarado. Así, se puede llamar a *oregon* y *maine* durante todo el programa, pero al procedimiento *portland* sólo se le puede llamar desde dentro de *oregon*. Y hay algo que añadir a esto: sólo se puede hacer referencia a un nombre después de haberle declarado; por consiguiente, no se puede llamar a *maine* en el interior de *oregon*. Si tenemos montones de rutinas que se llamen unas a otras, ello puede suscitar problemas de ordenación; de ellos hablaremos más adelante.

Espacio de memoria para las variables

En conexión con el ámbito local, hay un mecanismo relacionado con él, el cual se llama *tiempo o duración de vida local* (*local lifetime*). Si una variable es local de una rutina R, el PASCAL sólo asigna espacio de memoria para dicha variable cuando se llama a R; al producirse el retorno de esta llamada a R, el PASCAL libera el espacio de memoria, y la variable deja de existir. Mientras R se encuentre activa, podría llamar a otras rutinas, haciendo así que otras variables vengan y se vayan, pero esto no tiene efecto alguno sobre las variables locales de R. Si se llama a R una segunda vez, el proceso se repite; se asigna de nuevo espacio de memoria para sus variables y, al producirse el retorno desde R, dicho espacio se libera de nuevo. Obsérvese que la variable muere cuando se libera su espacio de memoria. Si la variable renace más tarde como resultado de otra llamada, tiene un valor indefinido. (El PASCAL no tiene, como ciertos BASIC no estándar, un sistema para inicializar todas las variables a cero.)

Refiriendo estas reglas al programa de nuestro ejemplo, inicialmente se asigna espacio de memoria a las variables *vermont* y *ohio* declaradas en el programa principal. Estas permanecen en existencia durante toda la ejecución del programa, siendo por tanto como las variables del BASIC. Supongamos que el programa principal llama a *maine*. En el mismo momento, nacen las variables *lincoln, bangor, portland* y *augusta*. (Las dos primeras son parámetros, y las otras dos son variables locales, pero todas estas cuatrillizas nacen al mismo tiempo.) Si posteriormente *maine* llama a *oregon,* nacen entonces *lincoln* (que no tiene ninguna relación con el *lincoln* anterior), *eugene, salem* y *foster*. Si, a su vez, *oregon* llama a su propio procedimiento local *portland,* nacerán entonces *fremont, lombard, union,* y nuevas encarnaciones de *vermont* y *foster*. Las variables que existen en este punto son las siguientes:

**vermont* *ohio*	pertenecientes al programa principal

*_lincoln_ *_bangor_ *_portland_ *_augusta_	pertenecientes a _maine_
lincoln _eugene_ _salem_ *_foster_	pertenecientes a _oregon_
fremont _lombard_ _union_ _vermont_ _foster_	pertenecientes a _portland_

Las variables marcadas con un asterisco se hallan en suspenso, o en hibernación. No se puede hacer referencia a ellas porque no se hallan actualmente dentro del ámbito. (Recuérdese que las variables pueden estar fuera del ámbito por dos razones diferentes: pueden estar en suspenso por precedencia de otra local, como _vermont_ y _foster,_ o pueden pertenecer a un procedimiento separado y no solapado, como las variables de _maine_). Cuando las variables en suspenso o hibernación entran de nuevo en el ámbito, como resultado del retorno de una rutina, vuelven a la vida sin haber sido afectadas por su sueño.

Las variables que se hallen dentro del ámbito en cualquier momento están determinadas únicamente por la disposición del programa. Por otra parte, las variables que se hallen activas están determinadas por el comportamiento dinámico de aquél, y particularmente por las rutinas que estén activas cuando se llama a una rutina dada. (En nuestro ejemplo, resulta que las variables _maine_ están activas cuando se llama a _oregon_.) Sin embargo, se puede garantizar que todas las variables incluidas en el ámbito están activas..., si no fuera así, la programación sería una verdadera pesadilla.

Siempre, tras los acontecimientos felices, vienen los funerales. Cuando se produce el retorno desde _portland,_ sus cinco variables locales mueren. Las _vermont_ y _foster_ originales, que antes se hallaban en suspenso, salen de su hibernación. Cuando se produce el retorno de _oregon,_ mueren sus cuatro variables, y las únicas que quedan para asistir al funeral son:

vermont _ohio_	pertenecientes al programa principal
lincoln _bangor_ _portland_ _augusta_	pertenecientes a _maine_

Todas éstas están dentro del ámbito. Si _maine_ llama a _oregon_ otra vez, renacen _lincoln, eugene, salem_ y _foster_. (Es muy probable que ocupen los mis-

mos espacios de memoria que tuvieron anteriormente, pero esto no se puede garantizar, de modo que sus valores iniciales son indefinidos.) Al producirse el retorno desde *oregon,* las nuevas encarnaciones de estas variables mueren de nuevo.

Finalmente, se producirá el retorno desde *maine,* matando a sus variables locales. Las únicas que podrán llevar luto por ellas serán las variables indestructibles *vermont* y *ohio.*

Diferencias con respecto a la asignación permanente de memoria

Si conservas una visión BASIC del mundo, es decir, si piensas que todas las variables excepto los parámetros han de tener espacio de memoria reservado permanentemente para ellas, es posible que tus programas en PASCAL funcionen aún, pero te perderás dos puntos importantes. Uno tiene que ver con la repetición, de la que hablaremos después, y el otro con las matrices o *arrays* grandes. Puede ocurrir que tengas algunas matrices grandes que sólo se necesiten durante un corto tiempo en la ejecución del programa. En tal caso, convierte cada matriz local en una rutina, y dispón tu programa de modo que llames a esa rutina cuando necesites la matriz o *array,* y se retorne cuando hayas terminado con ella. (En algunos casos, se puede hacer a varias matrices locales de una rutina, pero aquí hemos supuesto el caso de una sola.) El espacio de almacenamiento para la matriz está reservado solamente durante el tiempo de vida de la rutina; todo el resto del tiempo, el espacio de memoria está libre, y se puede utilizar para matrices locales de otras rutinas. En esta forma, el tamaño total de tus matrices puede ser mayor que la capacidad total de memoria que tengas; pero disponiéndolo todo en ámbitos que no se solapen, de modo que tus matrices no estén todas existentes al mismo tiempo, todavía podrás ejecutar tu programa.

Un lenguaje que dispone de los mecanismos arriba mencionados para la creación de ámbitos locales, y de duraciones o vidas locales de las variables, recibe el nombre de *lenguaje estructurado en bloques.* Ya mencionamos este nombre anteriormente, sin definirlo; la mayoría de los lenguajes modernos y muy utilizados (aunque, ciertamente, no todos) son estructurados en bloques.

La adaptación a la estructura en bloques

Muchos programadores de BASIC no obtienen del PASCAL todos los beneficios posibles, porque no están acostumbrados a pensar en términos de estructura de bloques. No caigas en esta trampa. Imita a la Sra. Buzz y haz el

esfuerzo preciso para dividir tu programa en rutinas separadas, ninguna demasiado larga, y asegurándote de que tus declaraciones estén tan localizadas como se pueda. Luego, cuando llegue el momento de hacer mantenimiento en tu programa, bendecirás al magnífico programador que lo escribió.

Para elegir un ejemplo concreto de ámbito local, considera el procedimiento *papelmojado* que definimos anteriormente en este mismo capítulo. La variable controlada de su bucle **for**, *índice,* se hizo local para *papelmojado.* (No supone derroche alguno de memoria el tener montones de variables locales separadas que actúen como variables controladas para bucles **for**; recuerda que sólo se adjudica espacio de memoria a las variables cuando se llama a su rutina.) Así, el procedimiento *papelmojado* es completamente autónomo.

Con un poco de suerte, alguna vez habrás tenido la siguiente experiencia. Has escrito un programa BASIC de la forma

```
100  FOR K = 1 TO N
     .
     .
     .
200  GOSUB 3000
     .
     .
     .
300 NEXT K
```

y te has encontrado con que no funcionaba. Al cabo de horas de frustración, has descubierto que la subrutina de 3000 utilizaba también a K en un bucle FOR, con el resultado de que la llamada a 3000 antes mencionada trastornaba al bucle FOR que la incluye.

Tal vez en aquellos momentos no te hayas considerado especialmente afortunado, pero ahora encuentras tu recompensa. En PASCAL siempre usarás una variable local para controlar a todos tus bucles **for**, y nunca te tropezarás otra vez con el mismo problema. En cambio, si no has aprendido la lección, llevarás indudablemente al PASCAL tus hábitos adquiridos en el BASIC, y más tarde sufrirás tu castigo.

Variables no locales

Si llegas a convencerte de que las variables locales son extremadamente valiosas, no hay duda de que, con el celo exagerado de todo neófito, tratarás de hacer un uso excesivo de ellas. De modo que hemos de advertirte que hay algunos sitios en los que las variables locales no son apropiadas. Uno de ellos quedará ilustrado si nos referimos a la función *leecubo* anteriormente utili-

zada, que utiliza una variable, *leecuenta*, para contar los números leídos. Si haces a *leecuenta* local de *leecubo*, aquélla morirá cada vez que retornes de *leecubo*, perdiendo así su valor. Por tanto, será obligado declarar a *leecuenta* en el programa principal o, si no, en alguna rutina que incluya a *leecubo* y que permanezca activa todo el tiempo que dure el funcionamiento del contador. También será necesario inicializar *leecuenta* a cero antes que se produzca la primera llamada a *leecubo*.

El principio general es que no se puede hacer local a una variable si se quiere que su valor se arrastre desde una llamada a la siguiente.

Repetición o recurrencia

Una rutina puede llamarse a sí misma. A esto, que es algo extremadamente útil, es a lo que se llama *repetición*, o *recurrencia*. Más adelante daremos ejemplos de ello.

También es posible tener un tipo más profundo de recurrencia, conocido como *repetición,* o *recurrencia mutua.* Consideremos dos procedimientos en un programa de ajedrez. Uno se llama *MejorJudadadeNegras,* y el otro *MejorJugadadeBlancas.* A cada uno de ellos se le pasa, como argumento suyo, una representación del estado del juego. La estrategia de *MejorJugadadeNegras* es proponer una jugada posible y llamar entonces a *MejorJugadadeBlancas* para averiguar cuál sería la respuesta a ella. El procedimiento *MejorJugadade-Blancas* hace exactamente lo contrario. Estos procedimientos, que se llaman el uno al otro, son un ejemplo de repetición o recurrencia mutua. La repetición mutua puede implicar a cualquier número de procedimientos, llamados hasta cualquier profundidad. Es claro que ha de haber alguna condición de parada o interrupción de la recurrencia, para evitar que ésta se vaya haciendo más y más profunda. Así, nuestros procedimientos de ajedrez podrían contemplar una profundidad de cuatro jugadas previstas, en cuyo punto se interrumpiría la repetición.

De hecho, nuestra función del *MáximoComúnDivisor* puede codificarse en una forma repetitiva. Esta versión recurrente, a la que hemos dado el nombre abreviado de *mcd,* se declara en la forma siguiente:

```
function mcd(número 1: poso0integer;número2:poso0integer):poso0integer;
(* Equivalente repetitivo de la función MáximoComúnDivisor *)
begin
   if número2 = 0 then
      mcd: = número1
   else
      mcd: = mcd(número2, número1 mod número2);
end; (* mcd *)
```

Esta versión tiene el mérito de funcionar cuando cualquiera de los dos argumentos es cero, así que hemos usado el tipo *poso0integer* en lugar del *enteropositivo*.

Para ser leales contigo, hemos de advertirte ahora que la recurrencia o repetición es uno de esos medios o facilidades que se describen paradójicamente como "muy fácil de usar cuando se sabe hacerlo". El saberlo lleva cierto tiempo para algunos, y si al echar un vistazo al *mcd* te sientes algo mareado o atemorizado por las alborotadas aguas que ves ante ti, te sugerimos que viajes en avión, volando adelante hasta llegar a la sección titulada "Referencia hacia adelante".

Al resto de los lectores, es decir, a los no atemorizados, se les puede explicar la forma en que funciona el *mcd* considerando el ejemplo de llamada

mcd(119, 98)

La base del algoritmo de Euclides es que el m.c.d. de 119 y 98 es el mismo de los dos números 98 y 119 *mod* 98 (es decir, 21); Si seguimos aplicando este proceso iremos obteniendo números cada vez más bajos, hasta que uno de ellos sea cero; en este momento, el otro número es la solución. Así, con nuestra función *mcd,*

mcd(119, 98) (1)

llama a

mcd(98, 21) (2)

que, a su vez, llama a

mcd(21, 14) (3)

que, a su vez, llama a

mcd(14, 7) (4)

que, a su vez, llama a

mcd(7, 0) (5)

La repetición se interrumpe ahora, porque *número*2 = 0 es verdadera. Así, la llamada más profunda de *mcd* devuelve el resultado 7 a la llamada (4), la cual devuelve el resultado 7 a la llamada (3), y así sucesivamente hasta que se devuelve 7 como resultado de la llamada externa.

Esta es la magia del algoritmo de Euclides. Funcionó perfectamente en la antigua Grecia, y todavía sigue haciéndolo en nuestros días.

La descripción repetitiva que el PASCAL hace del algoritmo es completamente natural. En realidad, pone de manifiesto que la pesada mano de Frank Round había intervenido en nuestro algoritmo original *MáximoComúnDivisor*. Este error original, basado en el BASIC, es más difícil de comprender que la descripción repetitiva (una vez que uno está acostrumbrado a la repetición).

Las variables locales y la repetición

Como segundo ejemplo de recurrencia o repetición, consideremos la función

```
function sumto0: integer;
var
   número: integer;
begin
   read(número);
   if número = 0 then
      sumto0:= 0
   else
      sumto0:= sumto0 + número;
end; (* sumto0 *)
```

Aquí, *sumto*0 es una función sin argumentos. Se la llama simplemente escribiendo su nombre. Cuando se mira a la línea

```
sumto:= sumto0 + número;
```

uno podría pensar que esto no es más que incrementar el valor de *sumto*0. Sin embargo, la sintaxis es bastante engañosa. (Recuérdese lo que dijimos anteriormente: que un nombre de función en el término izquierdo de una asignación es algo especial.) La segunda *sumto*0 es una llamada (repetitiva) a la función. Así pues, la línea significa "el resultado de la función se obtiene llamando repetitivamente a *sumto*0 y añadiendo el valor de *número*".

Antes que expliquemos lo que hace la función, trata de everiguarlo tú mismo. Considera la sentencia

```
writeln(sumto0);
```

cuando los datos suministrados para el *read* están formados por los números 13, 19 y 0.

La respuesta es que la función sigue leyendo datos hasta que se suministra un cero, y entrega como resultado la suma de los números sobre los que ha pasado. Equivale exactamente a

```
function sumto0: integer;
var
   número, suma: integer;
begin
   sum:= 0;
   repeat
      read(número);
      sum:=sum + número;
```

```
    until número = 0;
    sumto0: = sum;
  end; (* sumto0 *)
```

Tal vez te sorprenda que una operación tan mundana se pueda expresar usando la repetición y, más aún, que la versión repetitiva sea más corta. Sin embargo, si hace sólo unos momentos estabas luchando para tratar de averiguar lo que hace, tal vez no creas que la versión repetitiva sea más legible. Es más, puede que ni siquiera hayas podido llegar a determinar lo que hace.

Por si fuera así, explicaremos cómo trabaja esta función. Cuando se la llama por primera vez, la función lee el valor 13 y lo pone en *número*. Como *número* no es cero, la acción siguiente es la que viene después de **else.** Esto hace que se llame a *sumto*0 en forma repetitiva. Ahora viene el punto que verdaderamente nos interesa destacar en esta sección. Al tener lugar la llamada repetitiva, se asigna de nuevo espacio de memoria para *número*. La asignación anterior de *número* pasa a hibernación..., no podemos hacer referencia a ella, porque el nombre *número* siempre se refiere a la asignación más reciente. Esta segunda llamada a *sumto*0 pone a su *número* en 19, y el actual tiene el valor 0. (Tanto el valor externo como el segundo están en hibernación.) El tercer nivel halla ahora que *número* = 0, y retorna desde la llamada actual a *sumto*0, dando 0 como resultado de la función. En este punto se libera el tercer *número* y volvemos al segundo nivel. El segundo *número* pasa a ser el actual y, no afectado por su experiencia de haber estado algún tiempo en hibernación, tiene todavía el valor 19. Se toma de nuevo la sentencia

```
  sumto0: = sumto0 + número;
```

Recuérdese que la llamada repetitiva a *sumto*0 hizo que ésta quedara en suspenso. Ahora sabemos que el resultado de esta llamada es 0. Así pues, esta sentencia suma 0 a 19, y devuelve el resultado 19 como valor de la segunda llamada a *sumto*0. Estamos otra vez en una situación similar a la de la primera llamada, y sumamos 19 a 13, obteniendo 32 como resultado final. Así que el efecto de nuestra *writeln*(*sumto*0) original es la impresión del valor 32.

Todo el funcionamiento depende de que *número* sea una variable local. Si la declaramos fuera de la función, no se asignará de nuevo espacio de memoria cada vez que se llama a *sumto*0. En lugar de ello, lo que habrá será sólo una copia del *número*. Entonces, el resultado de *sumto*0 sería siempre 0, y no sería una función muy útil.

Si nuestra explicación anterior ha tenido éxito, te habrá convencido de que la repetición y las variables locales, trabajando hombro con hombro, constituyen una combinación muy potente. Incluso podrás creer al profesor Primple cuando mantiene que la repetición es el mecanismo fundamental de la programación. Sin embargo, tememos que muchos lectores puedan estar arrepintiéndose de no haberse ido con los que tomaron el avión, los cuales están tomando tierra precisamente ahora.

La referencia hacia adelante

Los grandes programas pueden contener cientos de rutinas, docenas de las cuales podrían estar declaradas al mismo nivel de anidamiento. Para hacer que un programa sea legible, ha de haber un ordenamiento coherente, de modo que un lector tenga probabilidades de hallar en la jungla una sentencia determinada. Un ordenamiento posible es la ordenación alfabética de los nombres de las rutinas. Sin embargo, esto nos lleva a un problema que ya hemos mencionado antes: al igual que cualquier otro objeto, el nombre de una rutina se ha de declarar antes de usarle. ¿Qué ocurre si el procedimiento *abeja* llama al procedimiento *zorro* o, peor todavía, si el procedimiento *zorro* llama también al *abeja* en forma mutuamente repetitiva? A fin de superar este problema, el PASCAL provee un mecanismo para hacer declaraciones adelantadas. Ello permite escribir, antes que *abeja,* la declaración

```
procedure zorro(pig: real);
    forward;
```

Esto cuenta como una declaración, en el sentido de que *zorro* está ahora dentro del ámbito y puede ser usado por otros procedimientos. Más tarde habrá que dar la declaración íntegra de *zorro*. Sin embargo, cuando lo hagas, habrás de omitir la lista de parámetros, porque ya la has mencionado una vez, y el PASCAL se molesta si se menciona dos veces. Así pues, podrías escribir

```
procedure zorro; (* pig: real *)
var
    (* ... *)
begin
    (* ... *)
end; (* zorro *)
```

Es una buena idea el repetir la lista de parámetros en un comentario, como hemos hecho más arriba, de manera que el programa siga siendo legible, a pesar de los intentos del PASCAL por impedirlo.

Similarmente, si *piggy* es una función que haya de declararse por adelantado, se puede escribir

```
function piggy (pig: real): integer;
    forward;
(*  .
    .
    .  *)
function piggy; (* (pig: real): integer *)
(* Sigue la declaración propiamente dicha *)
```

Cambio del valor de los argumentos

Hay una cuestión importante que hasta ahora hemos pasado por alto: si se cambia el valor de un parámetro, ¿cambia también el argumento correspondiente?

Consideremos el siguiente procedimiento:

```
procedure númerosredondos (x: integer);
(* Como veremos, éste es un procedimiento sin utilidad alguna *)
begin
    x:= x — x mod 10 (* restar de x el resto de x/10 *)
end; (* númerosredondos *)
```

La finalidad del sencillo procedimiento precedente es convertir a su parámetro *x* en el múltiplo más próximo de 10 que sea menor que *x*. Es para ayudar a aquellas personas ignorantes que no se fían realmente de los números y que se sienten más a gusto si los números difíciles como 67 y 63 se expresan en "números redondos" como 60. Podría tal vez discutirse que el 67 se debería redondear a 70, pero no importa.

Sin embargo, el procedimiento no consigue su propósito, porque la regla normal en PASCAL es que el parámetro sea una variable local de su procedimiento, y cualquier cambio que se haga en el parámetro no afecta al argumento correspondiente. Así, en la llamada

númerosredondos(*temperatura*);

el valor de *temperatura* no cambia. Lo que ocurre es:

1) se crea una variable local *x,* que toma el valor actual de *temperatura,* digamos 23;
2) *x* toma el valor 20;
3) al producirse el retorno desde *númerosredondos, x* muere, con el efecto neto de que la llamada a *númerosredondos* no ha servido absolutamente de nada.

En este ejemplo concreto, el objetivo deseado se puede conseguir escribiendo *númerosredondos* en forma de función, y no de procedimiento, pero no siempre será así.

Para conseguir en todos los casos la meta deseada, lo que hace falta es un mecanismo que ordene tratar a un parámetro exactamente como sinónimo de su correspondiente argumento, de modo que un cambio en uno ocasione automáticamente un cambio correspondiente en el otro. El PASCAL dispone precisamente de esa facilidad; consiste simplemente en escribir la palabra **var** delante de cada parámetro que se haya de tratar como sinónimo. En el caso de *númerosredondos,* su encabezamiento debería escribirse así:

```
procedure númerosredondos (var x: integer);
```

Los parámetros sinónimos se llaman parámetros *variables* (o parámetros **var**), y los no sinónimos se llaman parámetros *de valor.* El argumento correspondiente a un parámetro variable ha de ser del mismo tipo que el parámetro. Este es uno de los sitios en que no se pueden usar tipos de subrango compatibles, pero diferentes. Es más, el argumento ha de ser (cosa bastante natural) una variable. Así, la llamada

númerosredondos(*temperatura* + 6);

no sería correcta. El argumento puede ser, sin embargo, un elemento de una matriz o *array.* (Hablaremos de las matrices más adelante; por el momento, limítate a pensar de $a[k]$ en PASCAL como A(K) en BASIC.) Un ejemplo es

númerosredondos($a[k]$);

El efecto aquí es que se calcula el valor de k en el momento de la llamada a *númerosredondos* (supondremos que tiene el valor 6), y entonces se trata a su parámetro como sinónimo del elemento $a[6]$ de la matriz o *array.*

Aunque todavía no hemos llegado al estudio completo de las matrices, vale la pena mencionar aquí que pueden pasar como argumentos matrices enteras, y que en este caso el parámetro correspondiente es normalmente un parámetro variable. Esto sería aplicable, por ejemplo, a un procedimiento *invert,* que invirtiese una matriz o *array* suministrada como argumento.

Nuestros procedimientos *númerosredondos* e *invert* nos sirven para traer a colación una valiosa técnica de programación. Si un procedimiento tiene parámetros variables, puede ser autónomo, y, sin embargo, podrá todavía comunicar sus resultados al mundo exterior a él.

Las rutinas como parámetros

Muy de vez en cuando, es útil usar el nombre de una rutina como parámetro para otra. Por ejemplo, puedes desear llamar a un procedimiento

integrate(f);

en donde f es el nombre de una función. El PASCAL tiene un mecanismo para esto, pero el informe PASCAL difiere de la norma PASCAL, y, lo que es más, algunos compiladores difieren de ambos. Por tanto, si tienes necesidad de este tipo de facilidad, habrás de consultar tu manual local.

Construyendo bloques

Cerramos este capítulo hablando de un punto general relacionado con el estilo de programación. Este punto va dirigido a aquellos lectores que sólo ten-

gan experiencia de los mecanismos (carentes de parámetros) GOSUB y RETURN del BASIC mínimo y que, en consecuencia, tal vez no puedan apreciar inmediatamente todos los beneficios de su recién hallada libertad.

Cuando te veas enfrentado a un difícil problema de programación, es probable que lo resuelvas por medio de una combinación de métodos ''descendentes'' y ''ascendentes'' (*top down*, o de arriba abajo, y *bottom up*, o de abajo arriba). El método descendente, o de arriba abajo, consiste en comenzar con el problema global y dividirlo continuamente en subproblemas, hasta que estos subproblemas sean fáciles de programar, el método de refinamiento progresivo resume este proceso. El método ascendente, o de abajo arriba, supone la clasificación y separación de los detalles de bajo nivel y su encapsulación en rutinas. Estas rutinas se pueden emplear a continuación para atacar a los problemas del nivel de detalle inmediato superior, y así sucesivamente. El enfoque de abajo arriba sólo es adecuado cuando se sabe la dirección en la que se va.

Los estilos de programación varían, pero un enfoque típico consiste en usar el método descendente para reducir el problema a partes susceptibles de ser tratadas, y luego emplear el método ascendente para programar estas partes. La elección de las rutinas adecuadas en cada nivel de detalle es una de las habilidades o artes de la programación. A menudo se eligen mal la primera vez, y los programas serán toscos y desmañados. Al cabo de algún tiempo, tu experiencia te dirá que hay algo que no está bien. Entonces variarás las especificaciones de algunas de tus rutinas; con suerte, elegirás las que mejor vayan, y entonces la programación se convertirá en un sueño. Lo que has hecho es una cosa muy importante: has fabricado las herramientas adecuadas para el trabajo en cuestión.

Las herramientas

La razón por la que el hombre se considera superior a otros animales es porque usa herramientas. Las herramientas han sido la clave del progreso humano. Indudablemente, otros animales, cuando piensan en estas cosas, opinan lo contrario y se consideran a sí mismos superiores al hombre. Sin embargo, como no es probable que muchos de esos animales sean lectores nuestros, aquí seguiremos en la corriente de opinión de la superioridad humana.

Las herramientas tienen un valor inmenso en la programación, y un buen programador sabe, al mismo tiempo, construir buenas herramientas y utilizarlas diestramente. Léase un estudio completo de este tema en un libro de Kernighan & Plauger (1981), cuya lectura es un verdadero placer.

El crear un procedimiento o una función es una forma de construir una herramienta. Una vez creada para que realice una tarea en tu programa, puedes utilizar de nuevo la misma herramienta para ayudarte cada vez que surja o se repita una tarea igual en cualquier otro lugar del mismo. Es más, si la

herramienta en cuestión reúne unas características suficientemente generales, puedes incluirla en una biblioteca (caja de herramientas), para su utilización en otros programas.

Por todo ello, lo que aconsejamos a aquellos programadores que deseen superarse y destacar sobre los demás es lo siguiente: cuando tropecéis con problemas, construid una herramienta autónoma para ayudaros. De ese modo, podréis utilizarla de nuevo para vencer los mismos problemas siempre que os surjan en el futuro. El construir herramientas no es fácil, pero, como dijimos en la sección precedente, cuanto más practiquéis mejores llegaréis a ser.

Tan cuidadosa del tipo parece…

TENNYSON, in memoriam, *describiendo cómo preserva la naturaleza las diversas especies*

6
Más sobre tipos sencillos de datos

Clases de tipos

Todos los tipos de datos que hemos descrito hasta ahora han sido *tipos sencillos,* que tienen un sólo valor. En los capítulos que siguen consideraremos otras clases de tipos de datos; en particular, estudiaremos las matrices o *arrays,* que son colecciones de varios objetos. Al ver la expresión "tipo sencillo", podrías esperar que otros se llamaran "tipos complicados", pero no hay ningún lenguaje de programación que admita jamás que algo es complicado. La idea es que otros tipos son sencillos también, aunque no tanto como los así llamados. Esto se parece un poco a los detergentes para lavar, que vienen en paquetes de tamaños familiar, gigante y tambor. No existe ninguno de tamaño pequeño, individual.

Antes de pasar a los tipos no tan sencillos, completaremos en este capítulo nuestro examen de los tipos sencillos, presentando dos que hasta ahora habíamos omitido. Se trata de los tipos *Boolean* y *char.* El primero recibe su nombre del apellido del lógico George Boole (1815-1864), que goza de la impar distinción de que su nombre sea escrito miles de veces todos los días por programadores de todo el mundo. Sin duda, esta conmemoración es mejor que la de erguirse en efigie en una plaza urbana, siempre cubierto de palomas. Por consiguiente, no regatees a *Boolean* su mayúscula inicial cuando escribas programas en PASCAL, aun cuando es probable que tu compilador te perdone si escribes *boolean.*

Datos Booleanos

El tipo de datos Booleano contiene sólo dos valores: *verdadero* y *falso*. El tipo Booleano existe en forma efectiva en el BASIC, aunque es posible que no te hayas dado cuenta de ello. Las expresiones de relación tales como

```
X > Y
```

producen un resultado Booleano. En el PASCAL se puede asignar este tipo de expresión de relación a una variable Booleana, por ejemplo:

```
var
    xpositivo: Boolean;
    x: real;
begin
(*  .
    .
    .  *)
    xpositivo:= x > 0;
```

Aquí la variable toma el valor *verdadera* (*true*) si *x* es mayor de cero; en caso contrario, es *falsa* (*false*). Como dijimos en el capítulo 4, cuando usamos una sentencia **if** (y similarmente una sentencia **while**, etc.), en PASCAL no nos vemos limitados u obligados a escribir una expresión de relación, como hemos hecho hasta ahora, sino que podemos escribir cualquier expresión Booleana. Una expresión Booleana es igual que una expresión aritmética, sólo que, en vez de manejar números, manejamos los valores *verdadero* y *falso*. Los operandos de una expresión Booleana son expresiones relacionales, variables Booleanas o constantes Booleanas. Los operadores son **and, or** y **not;** sus significados (Y, O y NO) son obvios. A continuación se dan algunos ejemplos del uso de expresiones Booleanas:

```
var
    xpositivo, xoypositivo, domingo, lloviendo: Boolean;
    x, y, p, q: real;
begin
(*  .
    .
    .  *)
    if xpositivo then (* ... *);
    if not lloviendo then (* ...obsérvese que "not" toma un sólo operando *);
    while domingo and lloviendo do (* ... *);
    xoypositivo:= xpositivo or (y > 0);
    if (not lloviendo or domingo) and (p > q) then (* ... *);
```

Obsérvese que en una sucesión de sentencias tal como

```
x: = 15;
p: = x + 3;
xpositivo: = x > 0;
x: = x — 22;
```

la última sentencia, que cambia a *x,* no cambia el valor de *xpositivo* (que sigue siendo verdadero), por la misma razón que no cambia el valor de *p.*

Observa también que si hallas en tu programa algo como

```
if xpositivo = true then (* ... *);
```

debes suprimirlo, porque fue el taimado Frank Round el que lo puso. Lo de = *true* es completamente redundante.

La finalidad de las variables Booleanas debe ser evidente. Hacen que los programas sean más fáciles de leer y comprender. Los programas BASIC típicos están muchas veces oscuros y confundidos por las variables numéricas que adoptan sólo los valores 0 y 1, por lo que son realmente Booleanas. Tu programa BASIC puede contener, por ejemplo, una variable K5 que es 1 si se ha realizado una determinada inicialización, y 0 si no se ha hecho. Similarmente, una variable M puede ser 1 si han de imprimirse mensajes, y 0 en los demás casos. Parte del programa BASIC puede decir:

```
    .
    .
    .
500 IF K5 = 1 THEN 600
510 LET K5 = 1
    .  |
    .  }  realiza la inicialización
    .  |
600 ...
    .
    .
    .
800 IF K5 = 0 THEN 900
810 IF M = 0 THEN 900
820 PRINT "INICIALIZACION REALIZADA"
    .
    .
    .
900 ...
```

Esto se puede escribir en PASCAL (usando *incializado* en lugar de K5 y *enviomensajes* en lugar de M) en esta forma:

```
var
    inicializado, enviomensajes: Boolean;
begin
(*  .
    .
    .  *)
    if not inicializado then
    begin
        inicializado: = true;
        (*  ...realiza inicialización...  *)
    end;
(*  .
    .
    .  *)
    if inicializado and enviomensajes then
        writeln(''inicialización realizada'');
```

Datos de carácter

Una variable PASCAL del tipo *char* puede tomar como valor cualquier carácter aislado. No es, por consiguiente, como una variable de cadena o *string* en BASIC, cuyo valor consiste en una sucesión de varios caracteres, pero sin embargo se puede usar como bloque fundamental de construcción, a partir del cual se pueden hacer cadenas. Las cadenas (*strings*) del PASCAL se tratan en el capítulo siguiente.

Todo lenguaje de programación que soporte caracteres o *strings* tiene un *juego de caracteres* asociado. Este especifica el conjunto de caracteres permisibles y la petición de los caracteres individuales. En todo juego de caracteres las letras vienen en orden alfabético; así, invariablemente, 'B' es siempre mayor que 'A'. Sin embargo, los juegos de caracteres pueden diferir, por ejemplo, en cuanto a si '+' es mayor que 'A'; o, incluso, si 'a' es mayor que 'A'; por consiguiente, si nos fiamos de una forma de petición determinada, estamos cortejando al peligro. Los primeros sistemas ordenadores tenían frecuentemente juegos de caracteres muy limitados, basados en los equipos perforadores de tarjetas, pero un compilador y un sistema operativo modernos deben soportar suficientes caracteres para atender a la mayor parte de vuestras necesidades. Ello es bueno si la petición de caracteres está basada en una norma aceptada, tal como el código ASCII.

Una constante *char*, exactamente igual que una cadena o *string* de caracteres, va entre las comillas sencillas, es decir

'a'

Por tanto, al pasar del BASIC al PASCAL, tienes que devaluar todas las comillas al 50 por 100.

En las líneas que siguen se muestran algunos usos de una variable *char*:

```
var
    clavedeclase: char;
begin
(*  .
    .
    .  *)
    clavedeclase:= 'a'
    if clavedeclase = 'b' then (* ... *)
    case clavedeclase of (* char y case van juntos frecuentemente *)
        'a': (* ... *);
        'b': (* ... *);
        'c', 'd': (* ... *);
    end;
```

Operaciones con los tipos sencillos

Hemos visto ya los cuatro tipos sencillos incorporados en el PASCAL: *real, integer, Boolean y char.* Estos se pueden complementar con tipos definidos por el usuario; en el resto de este capítulo usaremos como ejemplo de tipo definido por el usuario

animal = (*ratón, perro, león*); (* *en orden creciente de fiereza* *)

Además, un subrango de cualquier tipo sencillo es también un tipo sencillo.

Con cada tipo de datos van asociados algunos operadores que se pueden aplicar a los objetos de dicho tipo. Estos operadores son:

real	*, /, +, —
integer	*, **div**, **mod**, +, —
Boolean	**and**, **or**, **not**
char	(ninguno)

La anotación de ''ninguno'' frente a *char* no significa que este tipo carezca de utilidad. Los operadores relacionales o de relación (por ejemplo, >, etc.), al ser polimórficos, se pueden aplicar a todos los tipos sencillos (y siempre dan un resultado Booleano); además, el operador de asignación, por ejemplo,

a := *b*;

se puede aplicar a cualquier tipo de datos (excepto a los archivos, véase el capítulo 9). En general, cuando un operador tenga dos operandos (es decir, para cualquiera de los operadores anteriores excepto **not**), estos dos operandos han de ser del mismo tipo de datos, aunque, como ya hemos dicho, los *integers* y los *reals* se pueden entremezclar bastante.

El PASCAL no nos permite hacer cosas excéntricas, como efectuar la operación lógica **and** con dos números reales (*reals*), o decir

(1/2) * *true*

El PASCAL no se ocupa de medias verdades.

Puedes construir fácilmente expresiones PASCAL ricas en diferentes operadores, como

a:= *b* > *c* **or not** *d* **and** *f*;

Hay un conjunto de reglas de precedencia para determinar el orden en que se realizan o ejecutan los operadores. Por ejemplo, las reglas dicen que la asignación anterior equivale a

a:= *b* > (*c* **or** ((**not** *d*) **and** *f*));

lo que tal vez te sorprenda. (El **not** se hace en primer lugar, luego el **and** y luego el **or**; los operadores relacionales tienen la precedencia más baja).

Sin embargo, si no estás seguro de la precedencia relativa de dos operadores, no consultes el informe PASCAL para ver cuáles son las reglas. Simplemente, pon en tu expresión los paréntesis que hagan falta para dejar bien claro lo que quieres decir. Como resultado de ello, los lectores de tu programa no tendrán que tratar de adivinar qué es lo que éste pretende hacer.

Juegos de valores ordenados

Todos los tipos sencillos están representados por un juego de valores ordenado. En la tabla que sigue se muestran los miembros primero, segundo y último del juego de valores asociado con algunos tipos sencillos de datos:

Tipo de datos	*Primer valor*	*Segundo valor*	*Ultimo valor*
integer	—*maxint*	—*maxint* + 1	*maxint*
Boolean	*false*	*true*	*true*
char	(definidos por el juego de caracteres)		
animal	*ratón*	*perro*	*león*
1..10	1	2	10

El tipo de datos *real* es excepcional. ¿Cuál es el número real que sigue a 2.0? La respuesta es que ello depende enteramente de la exactitud de la máquina que estés utilizando. A diferencia de la mayoría de nosotros, un compilador PASCAL puede prohibir aquellas preguntas a las que no sea capaz de dar una contestación satisfactoria; y, como veremos, se aprovecha bien de ello.

Los operadores relacionales del PASCAL utilizan estos juegos ordenados de valores. El PASCAL soporta también varias funciones incorporadas, además de aquellas con equivalentes en BASIC que ya hemos mencionado. Varias de ellas se refieren a los juegos ordenados de valores. Una bastante importante es

ord(*x*)

que da el ''número ordinal'' de *x* en el juego de valores definido por el tipo de datos de *x*. El primer valor del juego tiene el número ordinal 0, el segundo tiene el número ordinal 1, y así sucesivamente. Así, un ejemplo usando nuestro tipo *animal,* si la variable *bestia* estuviese establecida por el programa

```
var
   bestia: animal;
begin
(*  .
    .
    .  *)
    bestia: = perro;
```

entonces *ord*(*bestia*) sería 1, porque *perro* es el segundo valor en la sucesión ordenada de valores en el tipo de datos *animal.*

Si *x* es del tipo *char, ord*(*x*) da la posición de *x* en la ordenación definida por el juego de caracteres. Para hacer la traducción inversa a *ord,* hay una función *chr* que produce un resultado *char;* esta función es tal que

chr(*ord*(*c*)) es idéntico a *c*

para cualquier carácter *c*. Por tanto, la función *chr* es idéntica a la CHR$ que se encuentra en algunos BASIC, y, cuando se aplica a un argumento de carácter, el *ord* del PASCAL es como el CHR o el ASC del BASIC.

Finalmente, aunque no es posible sumar ni restar uno de, por ejemplo, una variable *char,* se puede conseguir el efecto equivalente con las funciones *succ* y *pred,* que significan sucesor y predecesor, respectivamente. La primera es tal que *succ*(*x*) es el sucesor de *x* en el tipo de datos definido por *x*. Así, *succ*(2) es 3, y, lo que nos viene más al caso, *succ*(*ratón*) es *perro,* dada la definición de animal expresada más arriba. La función *pred* hace lo contrario que la *succ.*

Si se pregunta por el *succ* o el *pred* de un número real, el PASCAL, como ya hemos advertido de antemano, declarará que la pregunta es improcedente.

El control de los bucles

En algunos ejemplos precedentes hemos venido usando *succ* y *pred* de una manera implícita. Ello es así porque **for ... to** emplea el *succ,* y **for ... downto** usa el *pred* para alterar la variable controlada. El resultado de esto es que la sentencia **for** es mucho más potente y tiene un carácter mucho más general de lo que su empleo con enteros *(integers)* muestra. Se puede decir

```
for bestia:= ratón to león do
```

en cuyo caso *bestia* tomará sucesivamente los valores *ratón, perro* y *león*. Similarmente,

```
for bestia:= león downto perro do
```

dará a *bestia* sucesivamente los valores *león* y *perro*. En un ejemplo más

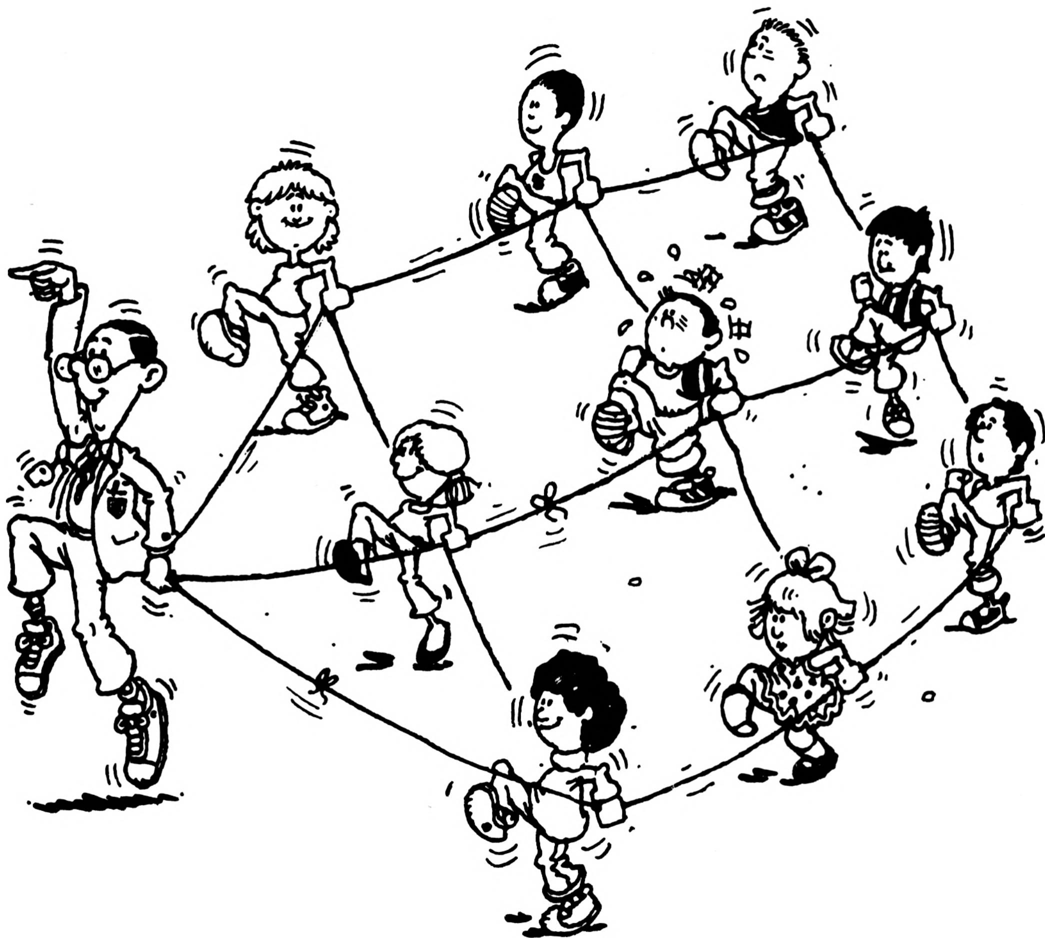
```
var
    ch: char;
begin
(*  .
    .
    .  *)
    for ch:= 'a' to 'z' do
        write(ch); (* write(ch) es como PRINT C$; en BASIC *)
    writeln; (* esto da salida a un carácter de cambio de línea *)
```

imprime la línea

abcdefghijklmnopqrstuvwxyz

siempre que las letras, de "a" a "z", se encuentren adyacentes en el juego de caracteres. En algunos juegos de caracteres, notablemente en el EBCDIC, hay caracteres adicionales, bastante peculiares, intercalados entre algunas parejas de letras.

En la mayoría de las aplicaciones se hallará amplio campo para el uso de bucles **for** de carácter general. No dejes que el Sr. 869704 te prive de la posibilidad de ser creativo al confinarte a los bucles **for** numéricos.



No te «encadenes» demasiado...

Consejo de RICARDO III para usuarios insatisfechos de PASCAL

7 Matrices *(arrays)* y cadenas *(strings)*

Conceptos básicos

Las referencias de matrices son básicamente similares en el BASIC y el PASCAL, con la salvedad de que este último encierra las listas de subíndices entre corchetes, en vez de hacerlo entre paréntesis; es decir, el A(J,K) del BASIC se convierte en el *a*[j,k] del PASCAL. El PASCAL no sólo permite las matrices de una o dos dimensiones, como el BASIC, sino también las de tres y más dimensiones.

En el PASCAL no existe equivalente directa de la sentencia DIM; en lugar de ello, las matrices o *arrays* se declaran en la sección **var,** exactamente igual que cualquier otra variable. Para lograr el efecto equivalente al de

DIM A(10), B(5,6)

se dice

var
a: **array** [0..10] **of** *real*;
b: **array** [0..5, 0..6] **of** *real*;

Si se desea que los límites inferiores sean 1 en lugar de 0, se puede cambiar por 1 el 0 correspondiente en la declaración anterior. En realidad, se puede, si se desea, hacer que el límite inferior sea cualquier otro entero (*integer*), como 4, o incluso — 4.

Sin embargo, el uso de enteros como límites sólo nos da un atisbo de la generalidad del PASCAL. Tal vez esto te sorprenda, pero, cuando declares una matriz o *array* en PASCAL, tienes que dar un tipo de datos para especificar cada par de límites. En nuestros ejemplos anteriores usamos tipos que son subrangos del tipo *integer,* por ejemplo 0..10. El tipo podría haber sido igualmente uno definido por el usuario, el tipo *char* o el *Boolean,* o un subrango de cualquiera de ellos. (El único tipo sencillo que no se permite es el *real.*) No dejes que el Sr. 869704 te engañe haciendo que todos los subíndices de tus matrices sean enteros o *integers.*

Con cada matriz van asociados por lo menos dos tipos de datos. Son el *tipo de datos de componente,* que es el tipo de datos de los elementos de la matriz y el *tipo de datos de índice,* que es el tipo de datos de índice de la matriz.

Según esto, en

c: **array** [*'a'*..*'z'*] **of** *Boolean*;

el tipo de datos de los componentes es *Boolean,* y el tipo de datos de índice es un subrango de *char.* La elección del tipo para el índice no tiene relación alguna con el tipo de los componentes. Si la matriz o *array* tiene varias dimensiones, puede tener varios tipos de índice distintos, por ejemplo:

ecount: **array** [*'a'*..*'z'*, *Boolean*] **of** 0..1000;

Dada la declaración precedente, se puede acceder a los elementos de *ecount,* que desde luego son enteros en el margen de 0 a 1000, por medio de subíndices tales como

ecount [*'p'*, *false*]

Las matrices tales como ésta no son un capricho. Supongamos que tenemos el problema de contar, en un texto dado, por cada letra de la "a" a la "z":

1) El número de palabras que comienzan con la letra dada y terminan con la *"e".*
2) El número de palabras que comienzan con la letra dada y no terminan con la *"e".*

En tal caso, la matriz o *array ecount* es justamente la estructura de datos que necesitamos para nuestro contaje. Tomando como ejemplo la letra *"p",* el elemento

ecount[*'p'*, *true*]

mide el número de palabras que comienzan con *"p"* y terminan con *"e".* El elemento *false* correspondiente cuenta el número de las que no acaban en *"e".* El

siguiente fragmento de programa muestra la forma en que se podría actualizar *ecount*:

```
var
    primercar: char; (* primer carácter de la palabra actual *)
    ultcar: char; (* último carácter de la palabra actual *)
    termicone:= Boolean;
    ecount: array['a'..'z', Boolean] of 0..1000;
begin
(*  .
    .
    .  *)
(* Supóngase que en este punto se han fijado primercar y ultcar *)
    termicone:= ultcar= 'e';
    ecount[primercar, termicone]:= ecount[primercar, termicone] + 1;
(*  .
    .
    .  *)
```

Piénsese en este ejemplo, y considérese cómo se podría programar lo mismo en BASIC. Ello debe poner de relieve varias de las lecciones que venimos tratando de enseñar en este libro.

Bucles con matrices

Dado que en el PASCAL normalmente se accede a los *arrays* o matrices dentro de bucles, es verdaderamente una suerte que las matrices y las sentencias **for** casen tan bien. Recuérdese que, lo mismo que el tipo de índice de una matriz no tiene por qué ser un entero o *integer,* tampoco necesita serlo la variable controlada para un **for**. Como ejemplo del uso de tipos de datos no enteros, la sentencia que sigue inicializa a cero todos los elementos de *ecount*.

```
for fila:= 'a' to 'z' do
begin
    ecount[fila, false]:= 0;
    ecount[fila, true]:= 0;
end;
```

en la que *fila* se declara como un variable del tipo *char*. De hecho, esta inicialización se puede escribir empleando un bucle anidado, que abarca los dos valores *false* y *true*. Esto se hace en la forma siguiente:

```
for fila:= 'a' to 'z' do
    for column:= false to true do
        ecount[fila, column]:= 0;
```

en la que *column* es una variable de tipo *Boolean.* Tal vez recuerdes que en PASCAL todos los tipos de datos son conjuntos ordenados y que, en el caso del *Boolean,* el *false* está definido un tanto arbitrariamente como menos que el *true,* como indicaba nuestra tabla del capítulo 6. (Si quieres una ayuda mnemotécnica, recuerda que si una sentencia es "menos que verdadera", es falsa.)

Un aparte acerca de la detección de errores

Como ejercicio sobre los efectos de los errores, vale la pena considerar el efecto que tendrá el escribir por error los límites *false* y *true* en el orden quivocado, es decir,

```
for fila:= 'a' to 'z' do
    for column:= true to false do
        ecount[fila, column]:= 0;
```

Los bucles nulos son completamente aceptables en PASCAL, al igual que en BASIC. Por tanto, el efecto del bucle precedente no sería inicializar a *ecount,* sino no hacer absolutamente nada, porque el bucle interno es nulo.

Afortunadamente, la mayoría de los compiladores PASCAL comprueban si existen *variables no asignadas,* es decir, el uso de variables a las que no se haya dado un valor. Por tanto, la primera vez que intentases usar un elemento de *ecount* recibirías un mensaje de error. Sin embargo, ésta suele ser una facilidad opcional, y muchos programadores precipitados la desconectan para que Perkins no intervenga y sus programas se ejecuten más rápidamente. En este caso, en el pecado llevarían la penitencia: en lugar de darles errores, sus programas correrían a gran velocidad produciendo respuestas completamente aleatorias, porque *ecount* empezaría con valores aleatorios.

Declaración de los tipos de índice

En la mayoría de los casos, el tipo de índice de una matriz o *array* será un tipo de subrango, tal como

'a'..'z'

o 1..10

Es útil seguir el consejo que dimos anteriormente, declarando tales tipos de subrango en la sección **type** del programa, y dándoles los nombres apropiados, por ejemplo:

```
type
    letras =  'a'..'z';
    vuelta =  1..10; (* suponiendo que el array representa tiempos para una
                         vuelta en una carrera de 10 vueltas a una pista *)
```

Las ventajas que esto tiene son de dos clases. En primer lugar, puede haber varias matrices o *arrays* relacionadas entre sí, todas las cuales compartan el mismo tipo de índice. Al declarar el tipo por separado, se hace más fácil el cambio. Por ejemplo, podríamos cambiar *vuelta* a 0..12, y todas las matrices que utilizaran *vuelta* cambiarían de tamaño conforme a ello. Mejor aún, podríamos ir un paso más adelante en la asignación de nombres y utilizar una **const** llamada *númerodevueltas* para representar nuestros 10 ó 12. Esta constante podría utilizarse en bucles **for** que se aplicasen a los *arrays* con el tipo de índice *lap*. Usando estas ideas podríamos tener un programa como el siguiente:

```
const
    númerodevueltas =  10;
type
    vueltas; =  1..númerodevueltas;
(* Llevaremos a la asignación de nombres un paso más adelente y daremos
    un nombre a nuestro tipo de array *)
    tiemposdevuelta  =  array [vueltas] of real;
var
    vueltasdeliebre:          (* tiempos de vuelta para la liebre *)
       tiemposdevuelta;
    vueltasdetortuga:         (* tiempos de vuelta para la tortuga *)
       tiemposdevuelta;
    cuentavueltas:            (* variable usada como índice para *)
       vueltas;               (* tiemposdevuelta *)
begin
    for cuentavueltas: =  1 to númerodevueltas do (* mete los tiempos de
                                                      vuelta *)
       read(vueltasdeliebre[cuentavueltas], vueltasdetortuga[cuentavueltas]);
(*   .
     .
     .  *)
```

La segunda ventaja que tiene el dar un nombre al tipo de índice es que se puede declarar que cualquier variable que se use como índice para una matriz o *array* es del tipo de índice en cuestión (véase *cuentavueltas* en el ejemplo precedente). Esto proporciona más seguridad. Sin embargo, nos lleva a hacer un comentario que nos duele anotar. Aunque Perkins es un tipo admirable y todo lo

que hace es con intención de ayudar, verdaderamente hay veces en que resulta un poco molesto. Una de estas ocasiones se produce cuando hay una variable cuyo tipo de subrango es justamente un poco pequeño. Más abajo mostramos un ejemplo de esto. El ejemplo ilustra también un valioso artificio de cálculo que se llama una *pila* o *stack*. Si nunca te has tropezado con pilas o *stacks*, no te preocupes; a pesar de ello, podrás comprender lo que pretendemos con nuestro ejemplo. Este es así:

```
const
    tamañopila = 100;
type
    margenpila = 1..tamañopila;
var
    pila:array[margenpila] of integer;
    partesuperiorpila:margenpila;
begin
    partesuperiorpila:= 1; (* vacía la pila inicialmente *)
(*    .
      .
      .  *)
(* Ahora se incrementa la partesuperiorpila, comprobando si se rebasa la
   capacidad (overflow) *)
    partesuperiorpila:= partesuperiorpila + 1;
    if partesuperiorpila > tamañopila then (* ...da mensaje error... *);
(*    .
      .
      .  *)
```

La cuestión es que si *partesuperiorpila* llega a ser igual a *tamañopila,* Perkins saltará en nuestra ayuda en cuanto intentemos incrementar *partesuperiorpila* y parará el programa allí y entonces. Desde luego, no se da cuenta de que la sentencia inmediata verifica si existe este error. Por tanto, deberíamos haber declarado *partesuperiorpila* como

```
    partesuperiorpila: 1..tamañopila + 1; (* ¡Pero no! *)
```

Desgraciadamente, esto no está permitido. El Sindicato de Operadores de Constantes de PASCAL, cuyos miembros trabajan altruistamente dentro de los compiladores de PASCAL tratando de hacer que los miserables programas que metemos en ellos lleguen a tener algún sentido, exigen que los límites de subrango sean constantes. Una expresión tal como 2 + 2 es demasiado, y sus miembros interrumpen el trabajo inmediatamente. Para aplacar al Sindicato, tendríamos que revisar el principio de nuestro programa para que dijera:

```
const
    tamañopila = 100;
    tamañoexcespila = 101; (* tampoco aquí permite el Sindicato que se
                              emplee una expresión *)
```

```
(* .
   .
   . *)
var
   partesuperiorpila: 1..tamañoexcespila;
```

Así pues, para volver a nuestro original, *algunas veces* es útil declarar que las variables empleadas como subíndices de *arrays* son del tipo de índice de *array*.

Sentencias MAT

> "Me estoy dando cuenta de que esta libre elección de los tipos de índice tal vez podría, en algunas circunstancias limitadas, servir quizá un poquitín de ayuda", dijo Bill, acumulando así sobre el PASCAL mayores alabanzas que en ninguna ocasión anterior, "pero, ¿cómo los usas con las sentencias MAT?"

Como sin duda sabes, hay varios BASIC que ofrecen las sentencias MAT, permitiendo operaciones tales como

```
MAT A = B+C
```

en la que A, B y C son matrices o *arrays*.

Para contestar a la pregunta de Bill —que, sin duda, sabía ya la respuesta cuando preguntó—, hay que decir que en el PASCAL no hay equivalente alguno de las sentencias MAT. Hay que escribir íntegramente todas las operaciones de matrices, usando sentencias **for** y demás.

La única excepción es que se puede asignar una matriz a otra, es decir, lo equivalente a

```
MAT A = B
```

en BASIC. Esto sólo se puede hacer cuando las dos matrices sean idénticas en cuanto al tipo de componentes y al (los) tipo(s) de índices. Y se consigue por medio de una sentencia ordinaria de asignación, por ejemplo:

```
a:= b;
```

Algunos compiladores de PASCAL tienen una regla muy estricta (a la que en el capítulo próximo se denomina de "equivalencia de nombre"), en virtud de la cual las matrices sólo son idénticas si están declaradas usando un identificador de tipo idéntico, como ilustra el ejemplo siguiente:

```
type
    recorridos = array [1..4] of integer; (* puntuaciones en un torneo de
                                                              golf *)
var
    a: recorridos;
    b: recorridos;
    c: array [1..4] of integer;
begin
(*  .
    .
    .  *)
    a:= b; (* es legal, porque a y b tienen tipos idénticos *)
(* ..pero c:= b; es ilegal, porque b y c no son de tipos idénticos *)
```

Esto refuerza nuestro consejo acerca de la conveniencia de declarar los tipos y de darles nombres. Al hacerlo, informas al compilador, y al lector de tu programa, acerca de cuáles son los objetos iguales y cúales los que difieren.

Matrices de matrices

No existe restricción alguna en cuanto al tipo de componentes de una matriz. Por tanto, al igual que puedes tener matrices de números reales, Booleanas, etc., puedes tener matrices de matrices. En efecto, puedes expresar cualquier matriz bidimensional como una matriz unidimensional cuyo tipo de componente es, a su vez, una matriz unidimensional. Así, una alternativa a la declaración de *ecount* es:

```
    newecount: array ['a'..'z'] of array [Boolean] of 0..1000;
(* en lugar de
    ecount: array ['a'..'z', Boolean] of 0..1000; *)
```

Entonces, para hacer referencia a los elementos individuales de *newecount* se usan dos juegos de corchetes, por ejemplo:

```
newecount['p'] [false]
```

Esto, por sí solo, no supone beneficio alguno (en realidad, probablemente consideres la idea un tanto retorcida), pero una matriz de matrices es útil en cuanto que te permite asignar submatrices o sub*arrays* de la matriz (llamadas "rebanadas" o *slices*), unas a otras, o a otras matrices del mismo tipo. Por ejemplo:

```
newecount['c']:= newecount['k'];
```

Si imaginas una matriz con sus filas y columnas, en las que el primer subíndice da la fila y el segundo la columna, la sentencia anterior asigna una fila de *newe-*

count a otra. En este caso las filas sólo contienen dos elementos, de modo que la asignación es equivalente, en la notación antigua, a

ecount[*'c', false*]: = *ecount*[*'k', false*];
ecount[*'c', true*]: = *ecount*[*'k', true*];

Nuestro ejemplo de matriz bidimensional se puede generalizar a cualquier número de dimensiones. Así, puedes tener matrices de matrices de matrices, o matrices unidimensionales de matrices bidimensionales, y así sucesivamente.

Un ejemplo

Para darte una idea un poco más completa del uso de las matrices o *arrays* en PASCAL, vamos a mostrarte ahora un programa algo más largo. En este ejemplo eludimos los conceptos complicados, como las matrices de matrices, y nos ceñimos a lo básico.

El programa ha de hacer los siguientes cálculos. Algunos datos consisten en el número de calorías que una persona que trata de adelgazar consume en cada día de la semana (empezando por el lunes) durante cinco semanas consecutivas. Cada semana tiene un ''día peor'', aquel en que el sujeto consume mayor número de calorías. El programa lleva la cuenta para el lunes, el martes, etc., del número de veces que dicho día de la semana ha sido el peor.

Las personas que tratan de adelgazar no sólo necesitan que se les avisen sus fallos; también necesitan estímulos. Aunque su ingestión de calorías pueda haber aumentado en una semana determinada, pueden tener la ligera compensación de haber consumido 20 calorías menos en un día concreto, en comparación con el mismo día de la semana anterior. Por tanto, el programa les informa de cuál ha sido el mayor avance (es decir, la mayor reducción en el consumo de calorías) en un día de la semana actual, en comparación con el día correspondiente de la semana anterior. Durante la primera semana, sin embargo, es evidente que no se puede imprimir esta cifra comparativa. Aquí tienes el programa:

```
program adelgazador(input, output);
type
    día = (lunes, martes, miércoles, jueves, viernes, sábado, domingo);
    ingestacal = 0..9999; (* ingestión calórica del día *)
    cuentascal = array [día] of ingestacal;
var
    calorías: cuentascal; (* cuentas calorías semana actual *)
    ultcal: cuentascal; (* cuentas calorías semana anterior *)
    cuentapeor: array [día] of integer;
    díapeor: día;
```

```
    hoy: día;
    avance: integer;
    mejoravance: integer;
    semana: integer;
    primerasemana: Boolean;
begin

(* **** 1: inicialización **** *)
    primerasemana:= true;
    for hoy:= lunes to domingo do (* inicializa cuentas día peor *)
        cuentapeor[hoy]:= 0;

(* **** 2; bucle principal para realizar cálculos semanales **** *)

for semana:= 1 to 5 do

begin
    for hoy:= lunes to domingo do (* lee los datos *)
        read(calorías[hoy]);
    díapeor:= lunes; (* valor inicial *)
    if not primerasemana then
        mejoravance:= últimascalorías[lunes] — calorías[lunes];
    for hoy:= martes to domingo do
    begin (* actualiza como sea necesario a díapeor y/o mejoravance *)
        if calorías[hoy] > calorías[díapeor] then
            díapeor:= hoy;
        if not primerasemana then
        begin (* compara con semana anterior *)
            avance:= últimascalorías[hoy] — calorías[hoy];
            if avance > mejoravance then
                mejoravance:= avance;
        end;
    end; (* para el bucle *)
    (* Al fin de semana:
        imprime díamejor y actualiza cuentas de díapeor *)
    if primerasemana then
        primerasemana:=false
    else
        writeln('Mejor avance en semana', semana, 'fue', mejoravance);
    cuentapeor[díapeor]:= cuentapeor[díapeor] + 1;
    últimascalorías:= calorías; (* una asignación de matriz *)
end; (* bucle semanal *)

(* **** 3: impresión final **** *)
    writeln;
    writeln('El número de veces que cada día ha sido el peor es el siguiente');
    for hoy:= lunes to domingo do
        writeln(cuentapeor[hoy]);
end;
```

Con estos datos

1596	1789	1500	2198	2709	1608	1763
1754	3408	2316	1908	1965	2076	1543
2409	987	1543	1765	1289	2341	1913
1720	1523	1482	1723	2178	1523	1478
3001	981	1234	2900	2709	2510	2301

el programa *adelgazador* produciría la salida

Mejor avance en la semana 2 *fue* 744
Mejor avance en la semana 3 *fue* 2421
Mejor avance en la semana 4 *fue* 818
Mejor avance en la semana 5 *fue* 542

El número de veces que cada día ha sido el peor es
2
1
0
0
2
0
0

Podrías introducir mejoras en la lógica de este programa. Tal como está, si dos días son igual de malos, el programa sólo cuenta el primero (cronológicamente) de ellos, dando así cierto sesgo a los resultados. También podrías mejorar la impresión, como explicaremos al final del capítulo 9.

Malas noticias

Si tú, lector, tienes muchos conocimientos de los lenguajes de programación, tenemos que darte algunas malas noticias. En cambio, si tu experiencia está limitada a los simples BASIC, puedes seguir sonriendo algún tiempo todavía, pues te parecerá que el impacto inicial de la tragedia se ha producido en un país lejano.

La mala noticia es que el PASCAL no provee mecanismo alguno para matrices o *arrays* de tamaño variable. Para los usuarios del BASIC, esto equivale a decir que no existe

```
DIM X(N)
```

en donde N sea una variable; en realidad, esto no debe constituir ninguna sorpresa. Sin embargo, en muchos otros lenguajes están permitidas las matrices de tamaño variable, y su ausencia en el PASCAL será una decepción para al-

gunos (aunque esto ya se les advirtió cuando comentamos anteriormente las reglas del Sindicato).

Una vez dada esta noticia, surge una cuestión más. ¿Es posible escribir un procedimiento cuyo parámetro sea una matriz o *array* de tamaño variable? Ciertamente, sería muy valioso para escribir un procedimiento que (por ejemplo) invirtiera una matriz de cualquier tamaño. Un procedimiento que pueda invertir sólo matrices de 5 × 5 es mucho menos útil.

Desgraciadamente, la respuesta a esa pregunta no es definitiva. El llamado "Nivel 1" de la norma del PASCAL (aunque no el informe sobre el PASCAL) permite que los parámetros matriciales sean de tamaño variable, pero harán falta varios años para que los compiladores existentes y en uso se modifiquen para adaptarlos a una nueva norma. Lo que es aún peor, existen por ahí muchos compiladores que permiten los parámetros matriciales de tamaño variable, pero lo hacen en una forma distinta de la forma de la norma. Por tanto, lo único que podemos ofrecerte es el consejo, un tanto vacuo, de que consultes tu manual PASCAL local, y que tengas cuidado con los problemas de portabilidad.

Otra restricción que tiene el PASCAL es que una función ha de producir como valor un tipo sencillo. Por tanto, no puede producir una matriz, aunque, como se verá en la próxima sección, puede asignar un valor a una matriz que se haya pasado como argumento.

Ejemplo de un parámetro matricial

El siguiente ejemplo muestra el uso de un parámetro matricial. El *array* o matriz es de tamaño fijo, estando constituida por diez elementos.

```
type
   vector10 = array [1..10] of integer;
var
   x: vector10;
   y: vector10;

procedure poneracero (var a: vector10);
var
   k: integer;
begin
   for k:= 1 to 10 do
       a[k]:= 0;
end; (* ponecacero *)

begin
   poneracero(x); (* ejemplo de llamada al procedimiento *)
   poneracero(y); (* otro ejemplo de llamada *)
(*  .
    .
    .  *)
```

Este ejemplo nos sirve para ilustrar otro punto. Una importante consecuencia de la regla relativa a los parámetros **var** es que, si quieres cambiar cualquiera de los elementos de la matriz pasada como argumento, o todos ellos, no tienes que olvidarte del **var.** Si nosotros nos hubiéramos olvidado de él para el parámetro *a* de nuestro procedimiento *poneracero,* los efectos habrían sido los siguientes:

- se habría copiado la matriz argumento, digamos *x,* en una matriz local llamada *a;*
- se pondría a *a* a cero;
- al retorno de *poneracero, a* desaparecería;
- la matriz original *x* permanecería sin cambio; de modo que el resultado neto de la llamada al procedimiento habría sido no hacer absolutamente nada. (De hecho, este ejemplo reproduce como en un espejo nuestro ejemplo *númerosredondos* del capítulo 5.)

Matrices compactadas (*Packed arrays*)

Supongamos que posees dos armarios alternativos para guardar tu ropa, cada uno de los cuales contiene un número de cajones de tamaño normalizado. Un armario tiene cuatro de estos cajones de tamaño normalizado, y el otro tiene uno sólo.

Teniendo en cuenta que tus necesidades en materia de vestuario son más bien modestas, puedes introducir todas tus ropas, apretándolas, en el armario de un solo cajón. Será difícil sacar la ropa, pero en cambio queda ajustadita, compacta, y es fácil de trasladar como un solo conjunto o paquete. La alternativa es usar el armario de cuatro cajones, poniendo calcetines en el primero, camisas o blusas en el segundo, y así sucesivamente. Ahora será más fácil sacar la ropa, pero su almacenamiento será menos compacto: tres cuartos de cada cajón estarán vacíos. También será más difícil transportar toda la ropa de un lado a otro como conjunto.

Unas consideraciones exactamente análogas se pueden aplicar a la compactación de datos dentro del ordenador. Cuando en PASCAL declaramos una matriz o un registro (véase más adelante), podemos darle el atributo **packed** o compactado. Con esto informamos al compilador de que estamos más interesados en la compacidad que en la velocidad de acceso a los elementos individuales. El compilador puede hacer o no hacer caso de nuestra advertencia; ello dependerá en gran medida de las facilidades de compactación de datos de que disponga nuestro ordenador. En algunas máquinas el **packed** puede suponer ciertamente la diferencia entre poder o no poder meter los datos en la máquina, por caber o no caber en ella.

La única facilidad de lenguaje que se pierde con el *packing* o compactado es que no se puede usar un elemento individual de una matriz como parámetro **var** para una llamada a un procedimiento o a una función. Es perfectamente permisible pasar toda una matriz compactada, con tal que tanto el argumento como el parámetro se declaren como **packed.** Por consiguiente, podríamos haber cambiado el precedente ejemplo de *poneracero* declarando a *vector*10 en esta forma:

*vector*10 = **packed array** [1..10] **of** *integer*;

y el programa seguiría siendo válido. Tres posibles líneas orientativas para el uso de la compactación son:

- el máximo beneficio se consigue mediante la compactación de *arrays* o matrices de pequeños objetos, tales como datos *char,* datos Boolean, o pequeños subrangos de enteros o *integers.* En breve veremos que los caracteres compactados tienen una importancia especial en PASCAL;
- si se accede frecuentemente al *array* o matriz *como conjunto* (por ejemplo, asignándola a otra matriz o pasándola como un parámetro no **var**), eso favorece el compactado, ya que los datos compactos son más fáciles de mover o transportar de un lado a otro;
- si accedes frecuentemente a elementos individuales de la matriz, ello hace que tu compactado sea menos atractivo.

Un método bueno e indoloro para evaluar la utilidad del *packing* o compactado en tu máquina, consiste en deslizar a hurtadillas unos cuantos **packed** en un programa de un amigo. Si, al día siguiente, te comenta orgullosamente lo mucho que ha conseguido mejorar el funcionamiento de su programa, ya sabes que el compactado es bueno, y entonces puedes probar a emplearlo en forma selectiva en tus propios programas.

El PASCAL provee procedimientos incorporados, llamados *pack* y *unpack,* para copiar una matriz no compactada en otra que lo esté, y viceversa (véase el Apéndice A).

Los *strings* o cadenas en BASIC

Vamos a presentar ahora el tema de los datos en *strings* o cadenas. Las distintas implementaciones BASIC difieren grandemente en cuanto a las facilidades que ofrecen para el manejo de cadenas. Sin embargo, tienen una cosa en común: todas ellas son mejores que las del PASCAL. Resulta una especie

de impacto enterarse de que las facilidades de manipulación de cadenas que tiene el BASIC (aparentemente sencillas) aventajan bastante no sólo a las del PASCAL, sino también a las de la mayoría de los nuevos, brillantes y actualísimos lenguajes estudiados por el profesor Primple y sus colegas.

La mayoría de los BASIC soportan cadenas de longitud dinámica, es decir, cuya longitud puede variar desde cero hasta el tamaño admisible. Así, se puede decir

```
LET S$ = "GETS"
LET S$ = S$ & "MAS LARGO"
```

El operador "&" concatena el valor anterior de S$ con MAS LARGO, ajustando así a S$ a la longitud de MAS LARGO. El resultado es que la longitud de S$ ha aumentado. Muchos BASIC, incluso en los microordenadores menos potentes, permiten que las cadenas varíen en longitud desde cero a 255 caracteres. Tal vez nunca te hayas dado cuenta de que ésta es una característica de lujo.

Los *strings* o cadenas en PASCAL

En PASCAL, una variable de cadena se representa como una matriz compactada *char*. (Algunos BASIC adoptan un sistema parecido, y permiten que se acceda a las variables de cadena como si fuesen matrices de caracteres individuales sencillos.) La matriz o cadena ha de tener como límite inferior el 1. El equivalente en PASCAL de las expresiones BASIC

```
LET R$ = "sí"
LET N$ = "Pedro"
```

es

```
var
   contestación: packed array [1..2] of char;
   nombre: packed array [1..5] of char;
begin
   contestación: = 'sí';
   nombre: = 'Pedro';
```

Al declarar las variables de cadena en esta forma, se obtienen todas las facilidades normalmente con las matrices o *arrays* en PASCAL. Entonces se podrá decir

```
contestación[j]: = nombre[3];
```

para asignar el tercer carácter de *nombre* al *j*ésimo carácter de *contestación*. Además, se consiguen tres facilidades adicionales que son exclusivas de las cadenas o *strings:*

1) se puede asignar una constante de cadena a una variable de cadena; las dos últimas líneas del ejemplo introductorio son muestras de esto;

2) se pueden aplicar operadores de relación a las variables y/o constantes de cadena o *string;*

3) se pueden sacar variables o constantes de cadena (pero, como veremos en el capítulo 9, no se pueden introducir más que carácter por carácter).

Un ejemplo que ilustra los puntos 2) y 3) es

if *nombre* <> 'Bruce' **then**
writeln(*nombre, 'no es australiano'*);

En PASCAL, las variables de cadena son de tamaño estático, es decir, su longitud es siempre la misma que la del *array* o matriz en que estén almacenadas. Por tanto, *contestación* ha de tener siempre un valor de exactamente dos caracteres, y *nombre* ha de tenerlo de exactamente cinco caracteres. Por consiguiente, no se puede decir

nombre: = *contestación;*

porque estaríamos tratando de dar a *nombre* un valor de dos caracteres. Similarmente, resulta que tanto Bill Mudd como el profesor Primple tienen razón cuando sentencian, desde sus puntos de vista diferentes, que no se puede comparar al BASIC con el PASCAL. La expresión de relación

'BASIC' > *'Pascal'*

nos dará un error con toda probabilidad, ya que las dos cadenas tienen longitudes diferentes. Si quitamos la "l" de PASCAL, podríamos probar

'BASIC' > *'Pasca'*

y, para disgusto de Bill, el resultado sería *false* porque el operador " > " utiliza el orden natural del diccionario. Lo mismo ocurriría si añadiéramos un espacio al final de BASIC. Por cierto que la relación

'BASIC ' > *'pascal'*

daría resultados indefinidos porque, como hemos dicho anteriormente, los juegos de caracteres pueden variar entre sí respecto a si las letras mayúsculas van o no delante de las minúsculas.

Como probablemente habrás notado ya, el Sindicato de Procesadores de Cadenas en PASCAL es todavía más fuerte que el Sindicato de Operadores de Constantes en PASCAL. Es tan fuerte, en efecto, que sus miembros apenas trabajan. Ciertamente, no podrás conseguir de ellos que rellenen una variable *string* o de cadena con espacios vacíos para completar la longitud exigida. Tal

vez pudieras convencerles para que rellenen una constante de cadena con el número adecuado de espacios. Por ejemplo, podrían aceptar

contestación: = *'no'*;

como sustituto de

contestación: = *'no '*;

Consulta los detalles en tu manual PASCAL local, ya que hay algunos compiladores que ofrecen ampliaciones no incluidas en el Sindicato.

Sin embargo, tendría que tratarse de un compilador muy ampliado o extendido para que ofreciera facilidades tales como la concatenación, u operadores que extraigan subcadenas de cadenas, como el LEFT$, el RIGHT$ y el SEG$ que se encuentran en algunos BASIC. Si existen, estas facilidades dan lugar a programas no portables, a menos que el PASCAL ampliado sea tan bueno que llegue a ser una norma *de facto,* cosa que ocurre algunas veces. La mayoría de los sistemas PASCAL no permiten ni siquiera definir una función que produzca un resultado en *string* o cadena, porque las funciones no pueden producir una cadena como resultado.

Si anteriormente has considerado que las restricciones del PASCAL sobre las matrices eran una tragedia ocurrida en una tierra lejana, ahora, tras ver el impacto que ello tiene sobre las cadenas o *strings,* pensarás que la tragedia se ha producido un poco más cerca de lo que creías.

El mismo profesor Primple ha encontrado algunos problemas en relación con las restricciones que el PASCAL impone en las cadenas.

> "Mi programa para analizar oraciones inglesas funciona bien", dijo, insistiendo en un tema familiar, "pero tiene una pequeña restricción, en cuanto que obliga a que todas las oraciones contengan diez palabras, y cada palabra ha de ser de cinco letras."

Los comentarios de Bill Mudd acerca de este aspecto del PASCAL no pueden compactarse en una cadena de longitud fija.

Superando las restricciones

Si al llegar aquí estás desesperado, tal vez te anime el saber que es posible burlar la intransigencia del Sindicato construyéndose uno sus propias cadenas o *strings* de longitud dinámica. Así que, en realidad, al profesor Primple le habría sido posible evitar las pequeñas restricciones de su programa.

Las cadenas o *strings* de longitud dinámica se pueden conseguir, por ejemplo, asociando dos variables con cada una de nuestras cadenas: una, un entero o *integer* que da su longitud actual, y la otra variable de cadena PASCAL cuya longitud sea la longitud máxima de nuestra cadena. Entonces uno puede escribir procedimientos para manipular estas cadenas o *strings*. Estos procedimientos sólo procesan los caracteres hasta la longitud actual; de los caracteres siguientes se hace caso omiso, aun cuando estén presentes. (Si quieres hacerte una idea de esta técnica, echa un vistazo a la pág. 168.) Cada cadena que se pase a estos procedimientos, se representa como un par de argumentos, que dan las dos variables asociadas con la cadena. Podríamos tener un procedimiento

assign(*longitud*1, *cadena*1, *longitud*2, *cadena*2);

que asigna una cadena a otra. Podría utilizarse este procedimiento en lugar de las facilidades incorporadas del PASCAL para la asignación de cadenas, y podrían escribirse procedimientos similares que tomaran el puesto de las restantes facilidades del PASCAL.

El resultado global de tus esfuerzos no sería bonito, y las constantes de cadena seguirían siendo un tedioso problema, pero por lo menos podrías rehuir las restricciones del Sindicato. El uso de registros, que se describe en el capítulo siguiente, te permite representar cada una de tus cadenas como un solo registro, en lugar del par de cantidades arriba descrito; esto contribuye a la concisión del programa.

Existen, como alternativa, algunas propuestas en la literatura concerniente a las cadenas o *strings* en PASCAL; véanse Bishop (1979) y Sale (1979). Estos representan las cadenas o *strings* como archivos (*files*), y tienes que hacer cosas raras, como rebobinar las cadenas antes de utilizarlas de nuevo. Sin embargo, cuando uno está desesperado no puede ser demasiado exigente o especial sobre el uso de argucias raras.

El registro de mi historial habla por sí mismo.

Afirmación popular

8
Registros

La declaración de un registro

El personal dedicado al aspecto comercial de los ordenadores ha venido utilizando el concepto de un *registro* desde finales de la década de los cincuenta. (*N. del T.*: Otra vez se divierte el autor en un juego de palabras, usando la palabra ''record'' en su doble sentido, como ''registro'' y como ''marca batida''.) El resto de la gente ha necesitado bastante tiempo para ponerse a su altura, y el PASCAL es uno de los primeros lenguajes que han popularizado los registros en campos más amplios.

Se utiliza un registro (*record*) para construir un nuevo tipo de datos que esté formado por una colección de otros tipos de datos (que pueden ser, y usualmente lo son, distintos entre sí).

El ejemplo siguiente muestra la forma en que se declaran y se usan los registros.

```
type
    solicitante =
        record
            peso: 1..30; (* peso en stones, medida inglesa equivalente a
                                14 libras *)
            edad: 0..150;
            legustaelPascal: Boolean (* Primple dice que se omita este
                                          punto y coma *);
        end;
var
    Primple: solicitante;
    Mudd: solicitante;
    solicitanteactual: solicitante;
```

Aquí el registro *solicitante* consta de tres *campos*: la edad y el peso del solicitante, y la Booleana *legustaelPascal,* que es verdadera o *true* si al solicitante le gusta el PASCAL. (Para beneficio de los lectores formados en esos países que no utilizan unidades tales como cadenas, varas, pértigas y perchas, y que probablemente tampoco juegan al cricket, diremos que nuestra unidad de peso es un *stone* (o piedra). Un *stone* es catorce libras. Una libra es, desde luego, el peso de un décimo de un galón imperial de agua. Un galón es...)

La única operación que se puede realizar con una variable que es un registro es asignarla a otra variable, que haya sido declarada como un registro idéntico. Así, podemos decir

```
solicitanteactual:= Primple;
```

o, esforzando mucho la imaginación,

```
Primple:= Mudd;
```

No existen constantes del tipo registro o *record,* del mismo modo que no las hay del tipo matriz (si se exceptúan las cadenas o *strings*).

Tipos estructurados

Tanto los registros o *records* como las matrices o *arrays* se llaman en PASCAL *tipos estructurados,* y tienen muchas cualidades comunes. Acabamos de ver que no es mucho lo que se puede hacer con los registros *como conjunto,* lo mismo que tampoco se puede hacer con las matrices o *arrays.* En lugar de ello, la mayoría de las operaciones se realizan sobre sus elementos individuales. A los elementos de las matrices accedemos por medio de subíndices; en cambio, a los campos de los registros accedemos por medio de los *identifica-*

dores de campo. Los identificadores de campo son los nombres que se escriben delante de los componentes del registro, es decir, en nuestro ejemplo, *peso, edad y legustaelPascal.* Desde luego, en un solo registro no puede haber dos campos con el mismo nombre, por ejemplo, dos *edades.* Sin embargo, sí que se puede usar el mismo identificador de campo en dos registros separados. De este modo, pueden tener el campo *edad* otros registros, además del de *solicitantes.*

Cuando queremos acceder a un campo de un registro, escribimos un punto después del nombre de la variable del registro, y a continuación el identificador de campo, por ejemplo:

```
Primple.edad:= 45;
Mudd.legustaelPascal:= false;
if cumpleaños then
    Primple.edad:= Primple.edad + 1;
```

Como ejemplo adicional, la sentencia de asignación

```
solicitanteactual:= Mudd;
```

equivale exactamente a

```
solicitanteactual.peso:= Mudd.peso;
solicitanteactual.edad:= Mudd.edad;
solicitanteactual.legustaelPascal:= Mudd.legustaelPascal;
```

Puede verse que la notación para acceder a los campos de los registros es ligeramente diferente de la empleada para acceder a los elementos de matrices, a saber:

variableregistro.campo en lugar de *variabledematriz*[*subíndice*]

Esta diferencia es razonable; deja ver claramente a un lector del programa si se está accediendo a un registro o a una matriz. No hace falta consultar la declaración.

Al igual que los elementos de matrices, los campos de los registros pueden utilizarse como variables ordinarias. Igualmente, como las matrices o *arrays,* los registros pueden ser compactados (**packed**), con ventajas potenciales similares.

Estructuración de datos

Es posible que hayas notado cierta semejanza entre el **begin** y el **end** empleados para agrupar sentencias, y el **record** y el **end** usados para agrupar datos. Esto nos da la clave del uso fundamental de los registros: ayudar a la Sra. Buzz a dar una estructura a los datos, lo mismo que el **begin** y el **end** (en

unión de otras construcciones) le ayudan a dar una estructura a las sentencias dentro de un programa. Ambas cosas son igualmente importantes. El título de la obra de Wirth *Algoritmos + estructuras de datos = programas* (1976) lo dice todo.

De la misma manera que se pueden anidar **begin** y **end,** podemos anidar registros. Para demostrarlo, vamos a modificar nuestro registro de *solicitantes* de modo que contenga un campo de fecha de nacimiento en lugar de un registro de edad. Algunos posibles patronos piden en sus impresos de solicitud de empleo los dos datos, edad y fecha de nacimiento, y eliminan a aquellos solicitantes cuya edad no concuerda con su fecha de nacimiento; se dice que los licenciados en informática son de los que más suelen fallar en esta prueba. Como no queremos ser demasiado duros con los licenciados en informática, prescindiremos del campo redundante de la edad. La fecha de nacimiento constituirá por sí sola un registro, formado por un día, un mes y un año. Entonces podremos declarar nuestro registro modificado en esta forma:

```
type
  solicitantea =
    record
      peso: 1..30;
      fechanacimiento:
        record
          día: 1..31;
          mes: (Ene, Feb, Mar, Abr, May, Jun, Jul, Ago, Sep,
                Oct, Nov, Dic);
          año: 1850..2000;
        end;
      legustaelPascal: Boolean;
    end;
```

Lo mismo podríamos haber escrito lo que antecede en esta forma:

```
type
  fecha =
    record
      día: 1..31;
      mes: (Ene, Feb, Mar, Abr, May, Jun, Jul, Ago, Sep, Oct, Nov,
            Dic);
      año: 1850..2000;
    end;
  solicitanteb =
    record
      peso: 1..30;
      fechanacimiento: fecha;
      legustaelPascal: Boolean;
    end;
```

La ventaja de los registros anidados debe ser evidente, ya que son análogos a otras estructuras anidadas. Su ventaja consiste en que es posible tratar como una unidad a un subregistro, como *fechanacimiento,* incluido dentro de un registro más grande.

Para acceder a un campo que se halle anidado a *n* niveles de profundidad en un registro, deben especificarse todos los *n* identificadores de campo que son necesarios para "hallar" el campo, comenzando por el registro más externo, por ejemplo:

Primple.fechanacimiento.mes: = *Feb*;

El ejemplo precedente es aplicable independientemente de si el registro ha sido declarado en la forma usada para *solicitantea* o en la de *solicitanteb.* Los identificadores de campos de los registros internos no han de ser iguales que los de los externos, pero, a pesar de ello, seguimos estando obligados a escribir todos los *n* identificadores.

Facilidades para el anidamiento

Hace algunos años, una emotiva frase publicitaria que trataba de animar a los británicos para que emigrasen a Australia, decía "En Australia, puedes". (El anuncio no registraba las frases correspondientes que podrían circular dentro de Australia, diciendo lo que podían hacer los británicos.)

Si deseas adoptar el estilo de programación de Bill, un eslogan equivalente a éste y relativo al PASCAL podría ser: "En PASCAL, no puedes". Ya comentamos, en el capítulo 2, la disciplina que el PASCAL impone.

A pesar de todo, existe un ámbito en el que el eslogan es, decididamente, "En PASCAL, puedes". Este ámbito es el del anidamiento. Efectivamente, casi siempre, y en forma invariable, está permitido llevar el anidamiento hasta cualquier profundidad que uno pueda razonablemente desear, y se puede anidar cualquier cosa dentro de cualquier otra. Por tanto, si preguntas si una facilidad *x* está permitida dentro de un registro, la respuesta será afirmativa. En consecuencia, podemos tener matrices de registros, o registros que contengan matrices, o podemos combinar ambas cosas como en el ejemplo que sigue:

```
array26 = array [2..6] of real;
profnido = array [1..5] of
    record
        a: array26;
        b: array [3..7] of
          record
                c: real;
                d: array ['a'..'e'] of Boolean;
          end;
        z: integer;
    end;
```

A tal estructura se podría acceder en la forma siguiente:

```
var
    x: profnido;
    yarray: array26;
begin
(*  .
    .
    .  *)
    x[1].b[3].d['c']:= true;
    yarray:= x[2].a; (* referencia a una matriz completa dentro de un
                        registro *)
```

Una opinión alternativa

"Bueno, todo este rollo sobre la estructuración está muy bien", dijo Bill, "pero, ¿para qué sirven estos registros? Todo lo que puedes hacer con ellos es asignarlos a otros registros. Me parece que son un caso más de acumulación de morralla redundante en los programas."

Bill está escribiendo también un libro. Lo titula *Al BASIC desde el PASCAL*. Espera que sea el mayor éxito de ventas de todos los tiempos, y fue tan amable que nos enseñó un primer borrador. En la primera página aparece este conmovedor consejo:

"¿Tienes problemas para recordar todos los tipos de datos del PASCAL? ¿Tropiezas con dificultades para decidir cuál de ellos usar? ¿Te preocupa que furtivos espías académicos puedan leer tus declaraciones?"

"Olvídate de todo ello. El mundo está formado solamente de números y de cadenas, además de matrices de cualquiera de ellos. Cualquier problema real se puede reducir siempre a estos tipos de datos, con tal que le des vueltas durante el tiempo suficiente. No te preocupes tampoco por los espías furtivos; lo más probable es que nadie más que tú sea capaz de averiguar lo que hacen tus variables, así que nadie te podrá criticar."

Un ejemplo

Sean cuales sean los méritos de los argumentos de Bill, es indudable que subestima bastante la utilidad de los registros "que sólo pueden asignarse a

otros registros''. Esto es porque un ejemplo de asignación está siendo utilizado como un argumento. Por tanto, puedes escribir procedimientos y funciones para manipular los registros que has definido.

Tomemos como ejemplo el registro

```
complejo =
    record
        partereal: real;
        imaginaria: real;
    end;
```

Este registro define un *número complejo,* que es un concepto muy utilizado en matemáticas e ingeniería y, por consiguiente, en los programas de ordenador. Los números complejos fueron inventados, con este mismo nombre, hace mucho tiempo. De haberse inventado ahora, y de acuerdo con algunas observaciones que ya hemos expresado, los encargados de las relaciones públicas les habrían dado el nombre de ''números sencillos'', o algo así, para distinguirlos de los números ''muy sencillos'' corrientes.

Para aquellos que nunca hayan tropezado con los números complejos o que, felizmente, los hayan olvidado, explicaremos que un número complejo es un par de números, a uno de los cuales se le llama la parte *real,* mientras que al otro se le conoce como la parte *imaginaria.*

Existe un conjunto de reglas para definir las operaciones aritméticas con números complejos. A continuación se muestran dos reglas: la notación (x, y) significa un número complejo con parte real x e imaginaria y:

Suma: $(a,b) + (c,d) = (a+b, c+d)$
Multiplicación: $(a,b) * (c,d) = (ac—bd, bc+ad)$

Dadas estas reglas, podrías pensar que sería fácil escribir un juego de funciones PASCAL para realizar aritmética de complejos. Sin embargo, ello no es posible, porque el resultado de una función PASCAL ha de ser un tipo de datos sencillo, mientras que nuestras necesidades demandan funciones que producen como resultado números (es decir, registros) complejos. De aquí que tengamos que formular las operaciones como procedimientos, y no como funciones.

Nuestro procedimiento para realizar sumas de complejos es:

```
procedure sumcomplej (var x: complejo; y, z: complejos);
(* Esto hace x:= y + z; en donde x, y y z son números complejos *)
begin
    x.partereal:= y.partereal + z.partereal;
    x.imaginaria:= y.imaginaria + z.imaginaria;
end; (* sumcomplej *)
```

Un procedimiento similar para hacer las multiplicaciones es:

```
procedure multcomplej (var x: complejo; y, z: complejos);
(* Esto hace a x:= y * z; en donde x, y y z son números complejos *)
begin
    x.partereal: = y.partereal * z. partereal — y. imaginaria * z. imaginaria;
    x.imaginaria: = y.imaginaria * z. partereal + y.partereal * z.imaginaria;
end; (* multcomplej *)
```

Dados estos procedimientos, una operación compleja tal como

```
c1: = c2 + c3 * c4;
```

se puede programar como

```
multcomplej(temp, c3, c4);
sumcomplej(c1, c2, temp);
```

en donde *temp* es una variable de tipo *complejo.*

Estos procedimientos recuerdan algo las rutinas MAT que se encuentran en algunos BASIC. La única diferencia es que aquí nosotros hemos definido nuestras propias operaciones; no tenemos que depender de que estén incorporadas en el lenguaje. (Podríamos haber definido nuestros propios equivalentes exactos de las sentencias MAT si el PASCAL permitiese usar como argumentos matrices o *arrays* de tamaño variable.)

Como forma alternativa para la escritura de nuestros procedimientos aritméticos para complejos, podríamos haber definido, en el programa principal, una variable compleja llamada *acumulador,* y haber revisado nuestros procedimientos para que dejaran el resultado de sus operaciones en este acumulador. Entonces los procedimientos no necesitarían el argumento *x,* y el estilo de programación se haría similar al de un lenguaje ensamblador típico (el lenguaje de bajo nivel para una máquina determinada), por ejemplo:

```
multcomplej(c3, c4);
sumcomplej (c2, acumulador); (* ahora el resultado está en el acumulador *)
```

Abstracciones

Nuestro ejemplo de los números complejos nos muestra un estilo de programación muy productivo. Lo que hacemos es definir nuestro propio tipo de datos, *complejo,* más un juego de procedimientos para trabajar sobre ese tipo de datos. Lo conseguido finalmente es casi tan bueno como si los *complejos*

hubieran estado incorporados en el PASCAL ya para empezar; la única sombra que lo empaña es que la sintaxis para el uso de nuestros procedimientos es bastante desmañada, aunque sin serlo de un modo intolerable.

Lo que hemos creado es una *abstracción de datos.* Se trata de una abstracción alejada de los tipos de datos incorporados en el PASCAL, y enfocada al problema que se está resolviendo. El PASCAL, aunque esté lejos de ser la herramienta perfecta, es una ayuda para conseguir esto. Obsérvese cómo los detalles referentes a la forma de realizar aritmética de complejos quedan confinados al interior de nuestros procedimientos. Por tanto, el programa principal queda libre de estos detalles de bajo nivel y, como resultado de ello, más fácil de entender. Esto es un ejemplo de ocultamiento de información.

El profesor Primple, cuyo nombre está muy asociado con la abstracción, cree que ésta es la más importante de las ayudas de programación.

Comprueba la validez de esta opinión suya haciendo algunas abstracciones por tu cuenta. Siempre que en un programa tengas una o más variables relacionadas entre sí (como una pila o *stack* y el indicador o puntero de la parte superior de la misma), agrupa estas variables formando un registro. Luego define algunos procedimientos y, si ello es apropiado, funciones para manipular los registros. ¿No se hace más fácil la programación? Lo que es más importante, ¿no es más fácil la lectura de tu programa? ¿No puede el lector, por ejemplo, separar fácilmente los conceptos generales de los detalles, y contemplar aisladamente los unos y los otros?

La sentencia *with* (con)

Volvamos a nuestros *solicitantes,* que llevan ya demasiado tiempo esperando.

Nuestra tarea consiste en escribir una función *evaluación,* que evalúa a los *solicitantes* de un determinado trabajo de programación. El que sea nombrado tiene que compartir una terminal, de modo que el solicitante ha de pesar lo suficiente para poder echar a otro a empujones. La función *evaluación* produce una puntuación que califica la adecuación del aspirante; cuanto más alta sea la puntuación, mejor. La función es como sigue:

```
function evaluación (persona: solicitante): integer;
(* evalúa la adecuación del solicitante de un trabajo *)
var
   puntuación: integer;
begin
   puntuación:= persona.peso; (* un punto por cada stone de peso *)
   if persona.legustaelPascal then
      puntuación:= puntuación + 50;
```

```
    if (persona.edad > 25) and (persona.edad < 45) then
      puntuación:= puntuación + 20;
    evaluación:= puntuación;
end; (* evaluación *)
```

(Perkins dice, con bastante razón, que esta función se podría mejorar definiendo un tipo subrango y usándolo en lugar de *integer* cada vez que éste se produce.)

A esta función se la puede llamar por código o clave, como

```
if evaluación(Primple) > evaluación(Mudd) then
  write('Primple')
else
  write('Mudd');
writeln(' consigue el trabajo');
```

Al codificar la función *evaluación,* fue bastante aburrido tener que escribir "persona" numerosas veces. Para ahorrar este aburrimiento, el PASCAL proporciona una especie de notación taquigráfica mediante la cual se puede decir "Dentro de este contexto, todos los campos se refieren a la variable de registro *x*". Esto se hace por medio de la sentencia **with** (con), que tiene una sintaxis (aunque no un significado) similar o parecida a la de la sentencia **while** (mientras). Usando **with,** la parte ejecutable de *evaluación* se puede escribir de nuevo en esta forma:

```
begin
   with persona do
   begin
      puntuación:= peso; (* un punto por cada stone de peso *)
      if legustaelPascal then
         puntuación:= puntuación + 50;
      if (edad > 25) and (edad < 45) then
         puntuación:= puntuación + 20;
   end; (* ámbito de with *)
   evaluación:= puntuación;
end; (* evaluación *)
```

De modo análogo, la parte ejecutable de *sumcomplej* se puede escribir

```
begin
   with x do
   begin
      partereal:= y.partereal + z.partereal;
      imaginaria:= y.imaginaria + z.imaginaria;
   end;
end; (* sumcomplej *)
```

aunque aquí el beneficio obtenido con el uso de **with** es marginal en el mejor de los casos. El ejemplo muestra, sin embargo, que dentro de **with** puede hacerse referencia a registros que no son el tema de **with.**

Existen formas más elaboradas de **with** que tú puedes probar cuando te hayas acostumbrado ya al caso sencillo. Y, desde luego, se pueden hacer anidamientos con **with.** Para detalles, consúltese el informe del PASCAL. No se olvide, sin embargo, que la idea de un **with** es hacer que el programa sea más fácil de leer y de escribir, de modo que uno complicado destruye su propia finalidad.

Recuérdense dos puntos en relación con el **with.** En primer lugar, se puede usar donde se quiera; su uso no está limitado a los procedimientos. En segundo lugar, se trata sencillamente de una especie de notación taquigráfica; no implica formación de bucles, ni nada parecido.

Efectivamente, un registro no lleva asociada ninguna facilidad para la formación de bucles. Esto contrasta con lo que ocurre con las matrices o *arrays,* en las que se puede hacer uso de una sentencia **for** para trazar bucles entre los elementos. Si lo piensas, te darás cuenta de que el realizar la misma operación en cada campo de un registro casi nunca es una cosa sensata, porque los campos son generalmente de tipos de datos distintos.

Dada la falta de un concepto de repetición en bucle, el orden de escritura de los campos de un registro es completamente el que uno quiera (excepto para los campos ''variantes'' que se describen en la próxima sección). Podríamos, por ejemplo, intercambiar el orden de *edad* y *peso* sin que ello afectase en absoluto a nuestros programas.

Registros con variantes

Uno de los problemas que presenta la programación en el mundo real es que los objetos no son uniformes. Así, si un registro de un programa describe a una persona, tenemos que una persona casada tiene un cónyuge, mientras que una soltera no lo tiene. Si la persona trabaja en una compañía y percibe un sueldo, los datos requeridos son diferentes de los de un jornalero; si la persona es extranjera, sus datos relativos a los impuestos serán diferentes.

Para tener todo esto en cuenta, el PASCAL permite que un registro contenga un *campo de variantes,* que ha de venir al final de aquél. El campo de variantes puede adoptar distintas formas, dependiendo de algún criterio especificado. Sin embargo, date por advertido: hasta al más fino y experimentado de los ''relaciones públicas'' le resultaría difícil convencerte de que la impresión inicial que produce esta característica no es excesivamente complicada. Por consiguiente, aprieta los dientes antes de pasar a lo que sigue.

Vamos a utilizar como ejemplo la existencia de piezas de repuesto en un almacén. Estas piezas encajan en dos categorías: algunas poseen un número entero, conocido como el ''número de pieza'', por el cual se las representa, en unión de una ''clave o código de clase'', que informa sobre la fortaleza de

la pieza, expresándola como *robusta, normal* o *delicada;* los componentes que no poseen número de pieza se representan por medio de una cadena de diez caracteres, que hace el oficio de ''nombre de pieza''.

El registro con variantes lleva asociado un campo *discriminador* (*discriminator*) para decir cuál de las posibilidades se ha de aplicar. En nuestro caso, el discriminador puede ser un *Boolean,* porque sólo hay dos posibilidades. Lo hemos llamado *tienenúmero*. La declaración del registro es:

```
type
   fortaleza = (robusta, normal, delicada);
   nombredepieza = packed array [1..10] of char;
   pieza =
      record
         coste: integer;
         (*  .
             .
             .  *)
         case tienenúmero: Boolean of
            true: (
               númeropieza: integer;
               códigoclase: fortaleza
                  );
            false: (
               nombrepieza: nombredepieza
                  );
      end;
```

Esta sintaxis, un tanto complicada, dice lo que más arriba hemos tratado de explicar en palabras. En detalle, dice que el último campo tiene varios casos (**case**) posibles. Puede argumentarse que el uso que el PASCAL hace aquí de la palabra **case** es una equivocación, ya que la sintaxis no es enteramente igual que la de la sentencia **case** descrita en el capítulo 4. (Sin embargo, vuestro compilador, que no olvida nada, espera a pesar de todo la palabra **case.**) La palabra **case** va seguida de una declaración del campo discriminador que se utilizará para seleccionar la variante. En el cuerpo de **case,** cada valor posible del discriminador va seguido por una lista (que puede ser vacía) de declaraciones de campos, encerrada entre paréntesis. Estas declaraciones de campos se escriben en la forma normal. Como en una lista de parámetros, no se escribe punto y coma después de la última declaración de la lista. Puede confundirnos un tanto el hecho de que **case** no lleve asociado un **end.** En lugar de ello, y dado que el campo de variantes siempre viene al final del registro, el **end** de éste sirve también para finalizar el **case.** Los campos que hemos declarado en el **case** anterior se pueden usar exactamente igual que si se hubieran declarado

```
tienenúmero: Boolean;
númeropieza: integer;
códigoclase: fortaleza;
nombrepieza: nombredepieza;
```

La única condición es, desde luego, que no se ha de intentar hacer referencia a un campo que no exista, como el *nombrepieza* de un componente para el que *tienenúmero* sea *true*.

A fin de evitar tales catástrofes, normalmente se usa una sentencia **if** o **case** para averiguar qué variante está presente. Por ejemplo:

```
var
    orden: pieza;
begin
(*  .
    .
    .  *)
    if orden.tienenúmero then
      if orden.códigoclase = delicada then
        writeln('Cuidado: es delicada');
```

Si escribimos, sin darnos cuenta,

```
if orden.tienenúmero then
    write (orden.nombrepieza);
```

o incluso sólo

```
write (orden.nombrepieza);
```

en un caso en el que tal campo es inexistente, el Admirable Perkins deberá cazarnos, ya que el campo *nombrepieza* no estará allí. Como siempre (o, por lo menos, casi siempre), aquí las actividades de Perkins son una gran ayuda para nosotros, porque impiden que los programas hagan disparates. (Sin embargo, es posible que tu compilador no haga la necesaria comprobación, en cuyo caso se perderá esta ventaja.)

Ventajas de las variantes

La ventaja más obvia de los registros con variantes es que economizan espacio de almacenamiento o memoria. Hay también otros dos aspectos igualmente importantes. El de la seguridad, ya mencionado, que impide que se haga referencia a campos inexistentes. Y también la cuestión de legibilidad, una vez que uno se ha acostumbrado a ellos: la descripción del registro facilita bastante información acerca de la forma en que el mismo está estructurado.

La restricción que exige que la parte variante de un registro venga al final del mismo, no es ninguna restricción en realidad. Esto es porque las variantes (naturalmente) pueden anidarse, de modo que podamos introducir varias variantes en un registro, si puedes tolerar la idea.

Una forma alternativa de las variantes

Finalmente, existe una forma alternativa para escribir las variantes, que deberá servir de recompensa a Bill Mudd, si nuestro amigo ha conseguido llegar hasta aquí. Esta alternativa nos permite omitir el campo discriminador, con el resultado de que el Admirable Perkins no podrá vigilar lo que hacemos. Se llama una ''unión indiscriminada''. Por ejemplo, podríamos decir que un campo es un número real, o un entero (*integer*) o una cadena de cuatro caracteres, y Bill podría establecer el campo, por ejemplo, como una cadena de caracteres y luego recuperarlo como entero o *integer,* sin que Perkins se entere. Si estás acostumbrado a programar en uno de los lenguajes llamados ''sin tipos'', como el BCPL (Richards & Whitby-Strevens, 1979), o incluso en lenguaje ensamblador, podrías apreciar las uniones indiscriminadas. Podrías definir un registro formado por un solo campo consistente en la unión indiscriminada de todos los tipos de datos que ocupen una sola palabra de tu máquina; esto te da el equivalente de una variable sin tipo.

Una matriz o *array* de estos registros es como la memoria de una máquina. La sintaxis es:

```
type
(* Utilización típica de una palabra en un ordenador de 32 bits *)
    Cuatrocasos = 1..4
    palabra =
        record
            case cuatrocasos of
                1: (
                    r: real
                   );
                2: (
                    i: integer
                   );
                3: (
                    str: packed array [1..4] of char
                   );
                4: (
                    B: Boolean
                   );
        end;
    store = array [1..1000] of palabra;
var
    w: palabra;
begin
    w.str:= 'abcd';
    wrinteln('Mi compilador almacena abcd como el número entero', w.i);
```

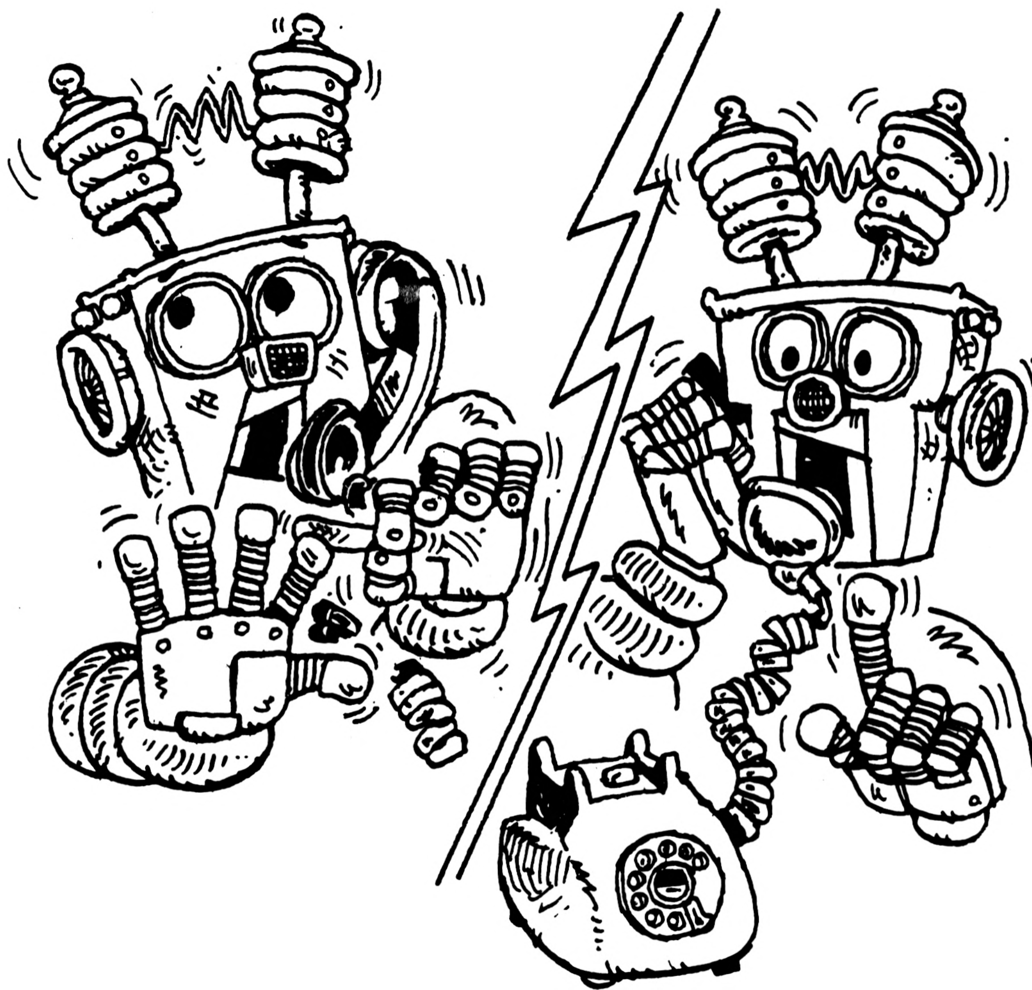
Obsérvese que el discriminador que sigue a **case** no tiene nombre, es simplemente un tipo.

Estas facilidades son útiles para los sucios trucos de programación que a Bill le gusta hacer, pero, aunque sea triste decirlo, casi todos tenemos que recurrir alguna vez a truquitos sucios para conseguir que nuestros programas se adapten a las coerciones del mundo real, o puedan interconectarse con los dispositivos de dicho mundo.

La elección de los números enteros 1, 2, 3 y 4 en el ejemplo precedente es completamente arbitraria, ya que en realidad nunca hemos establecido un discriminador. No obstante, si hemos de enumerar los casos, ésta es una secuencia natural que se puede emplear.

Equivalencia de nombre

Finalmente, vale la pena repetir un punto especializado, pero importante, que se expuso ya en el capítulo anterior. Algunos compiladores insisten en que, cuando se asigna una variable a otra, si las dos variables están declaradas separadamente, ambas han de tener un tipo de datos con un *nombre* idéntico. No es suficiente declarar que ambas son matrices o *arrays* del mismo tamaño, o registros con los mismos nombres de campos. (La única excepción se encuentra en los tipos simples, en los que es posible asignar tipos compatibles entre sí; por ejemplo, un subrango de, digamos, enteros o *integer,* se le puede asignar a un entero, y viceversa.) Esta estricta regla se conoce como *equivalencia de nombre,* y se aplica lo mismo a la asignación explícita que a la implícita de un argumento a un parámetro. (Véase un estudio más amplio en Welsh, Sneering & Hoare, 1981.) Si declaras todos tus tipos en la sección **type** del programa, dándoles nombres al hacerlo, estarás seguro. Lo mejor es dar nombres incluso a los tipos de datos de los campos en el seno de los registros. Así, nuestro *solicitanteb* está mejor que el *solicitantea,* porque el primero tiene un nombre (*fecha*) para el tipo de datos del campo *fechanacimiento.* Como resultado de ello, este campo puede ser asignado sin riesgo a cualquier otra variable del tipo *fecha.*



Uno de los mayores placeres de la vida es la conversación.

REV. SYDNEY SMITH

9
Entrada y salida

Frecuentemente se mantiene que la enseñanza moderna da una importancia excesiva a materias novedosas y de relumbrón a costa de otros conocimientos básicos. El PASCAL padece males parecidos. Es realmente fuerte en cuanto a estructuras de datos y cosas análogas, pero relativamente débil en lo que atañe a los campos de la lectura, escritura y aritmética. Su aritmética carece de un operador para la elevación a potencias y (omisión que sienten mucho los programadores comerciales) de operaciones con decimales. Su lectura y su escritura, es decir, su entrada y salida, parecen ideadas para favorecer la venta del libro *PASCAL a partir del BASIC* que prepara nuestro amigo Bill. No se trata tanto de que el sistema de entrada/salida del PASCAL disponga de pocas facilidades como, simplemente, de que hay algunas cosas fundamentales que es difícil o imposible hacer.

Categorías de archivos

Probablemente hayas pasado por la experiencia de tratar de imprimir un archivo y obtener como resultado un galimatías o una jerga extraña. Esto puede ser porque el contenido del archivo sea también un galimatías, pero fre-

cuentemente es debido a que el archivo está en memoria en una forma interna binaria que no es directamente imprimible.

Como sin duda sabes, la mayoría de los ordenadores almacenan los números en una codificación binaria interna que es diferente de la cadena de caracteres (cifras, comas decimales, etc.) que nosotros empleamos para representar los números. En realidad, la codificación interna de objetos tales como los números reales es muy elaborada, y es preciso mucho tiempo de ordenador para hacer la conversión entre (por ejemplo) 2.397E-3 y la codificación binaria equivalente.

De aquí que, para economizar tiempo en la conversión, muchos ordenadores tengan una facilidad para archivar información en una forma binaria que es, sobre poco más o menos, una imagen directa de la codificación contenida en la memoria o almacenamiento del ordenador. Estos son, entonces, los archivos cuya impresión produce un galimatías.

El PASCAL ofrece archivos de los dos tipos, de caracteres y binarios. Los primeros son los más corrientes, y se les conoce como *archivos de texto* o *textfiles.*

En BASIC, cuando se hace referencia a un archivo, se utiliza un número de canal. Si se omite este número de canal, se utiliza el archivo de entrada o salida previsto en el sistema para tomarlo por defecto u omisión (*default*). Así, se puede usar

```
PRINT X
```

para imprimir usando el archivo de salida por defecto, o

```
PRINT #3: X
```

para imprimir usando el archivo correspondiente al canal 3. Y para las sentencas INPUT se aplica un sistema equivalente.

El PASCAL tiene una filosofía semejante en cierto modo, sólo que en ella se ha desterrado la influencia del Sr. 869704. Por consiguiente, los números de canales están sustituidos por identificadores, que se denominan *variables de archivo* o, más simplemente, *archivos.* Estos identificadores se declaran en la forma normal en la sección **var** del programa. Si se desea, pueden ser locales de un procedimiento, bien sea como parámetros o como variables locales. La sintaxis para las declaraciones de variables de archivos se ilustra en los ejemplos que siguen:

```
var
    f1: file of char;
    f2: file of integer;
    f3: text;              (* véase la explicación más adelante *)
```

Un archivo es una secuencia o sucesión ordenada de componentes, todos los cuales son del mismo tipo de datos. Este tipo de datos se conoce como *tipo de componentes,* y puede ser cualquier cosa que uno quiera, aunque algunas

implementaciones del PASCAL imponen ciertas restricciones a los archivos de archivos.

Una declaración de archivo se parece algo a una de matriz, y más adelante volveremos sobre esta similitud. Si, como el *f*1 anterior, un archivo es de caracteres (*char*), entonces se trata de un archivo de texto (*textfile*), es decir, la clase de archivo con el que estamos más familiarizados. Si el tipo asociado con el archivo es cualquier otro menos el *char,* se tratará de un archivo binario. Como quiera que los archivos de texto son los más frecuentes, el PASCAL tiene un tipo de datos incorporado denominado *text,* lo cual significa **file of** *char.* Así, el *f*3 precedente es, al igual que el *f*1, un archivo de texto.

Los nombres *input* y *output* se consideran automáticamente como archivos de texto; corresponden a los archivos de entrada y salida por defecto, igual en gran medida que en el BASIC, y casi siempre corresponden a nuestra terminal. (Por consiguiente, el concepto de "archivo" se ha generalizado para incluir el material tecleado en la terminal, así como el material contenido en la memoria o almacén de reserva o de apoyo del ordenador. Los sistemas buenos ofrecen una completa independencia de dispositivos, como vimos en el capítulo 3.)

El uso de variables de archivo en las sentencias de entrada/salida se parece también al uso de canales numéricos en el BASIC. Se puede escribir una variable de archivo como primer argumento opcional o facultativo en los procedimientos *read* y *write* del PASCAL. Si se omite el argumento, se asumen *read* o *write* como sea apropiado. Así, las dos sentencias

```
read(x, y);
read(input, x, y);
```

son equivalentes, como lo son estas otras dos:

```
write(x, y + 6, 'pig');
write(output, x, y + 6, 'pig');
```

A continuación damos ejemplos del uso de las variables de archivo declaradas más arriba:

```
read(f1, x, y);
read(f2, x, y); (* también se pueden leer y escribir archivos binarios *)
write(f3, x, y);
```

Los tipos de datos en entrada/salida

Si un archivo es de texto (*textfile*), se pueden leer de él elementos de datos de entrada que sean de los tipos *integer, real* o *char.* Los elementos de datos

numéricos contenidos en el archivo han de estar separados entre sí por uno o más espacios. (El BASIC utiliza normalmente como separadores comas, en lugar de espacios.) Alternativamente, los elementos de datos de entrada se pueden poner en líneas separadas. Obviamente, cada elemento de datos ha de ser una constante del tipo apropiado. Si suministras los datos "xyz" cuando se necesite un número real, recibirás un aviso de error. Cuando se lee un número, se saltan todos los espacios y/o retornos de carro que le precedan, se toma el número, y se deja el archivo posicionado en el carácter que siga a dicho número. Cuando se lee un *char,* se toma un solo carácter, y un espacio cuenta como un carácter, al igual que cualquier otra cosa.

Como ejemplo de formatos de datos, si el programa contiene las líneas

```
var
    c: char;
    i: integer;
    r: real:
begin
    read(c, i, r);
```

la entrada de datos correspondiente puede ser

```
x 23 9.7
```

o

```
x
23
9.7
```

o

```
x 23
9.7
```

o, finalmente

```
x23
9.7
```

(Este último ejemplo muestra que no hace falta separador entre un carácter y un número.)

Obsérvese que la sentencia *read* hace la conversión automática desde la forma externa (por ejemplo, 9.7) a la binaria interna.

Para salida (*output*) a un archivo de texto, se pueden escribir (*write*) los mismos tipos de elementos que se pueden leer (*read*), es decir, *char, integer* o *real.* Hay, además, otros dos tipos de datos que se pueden escribir. La utili-

zación en ficheros de los Boolean es la respuesta del PASCAL a la memoria *write-only* (de sólo escritura); se pueden escribir Booleanos —que salen o resultan como *true* o *false*—, pero no se pueden leer. Algo similar ocurre con las cadenas o *strings*, pero se puede escribir con bastante sencillez un procedimiento para leer una cadena carácter por carácter.

Para aquellos archivos que no sean de texto, sólo se pueden leer y escribir elementos de datos que sean del mismo tipo de datos que el tipo de componentes del archivo.

Nombres de archivos externos

Hasta ahora, no vamos mal; los archivos en PASCAL son en gran medida como probablemente esperabas que fueran.

Los problemas empiezan cuando tratas de definir la correspondencia entre los nombres de archivo empleados en tu programa y los nombres reales de los archivos en tu sistema de archivo. Llamaremos a estos últimos *archivos reales*, o *actual files*, y a las variables de archivo correspondientes contenidas en tu programa, *archivos externos,* o *external files* (ya que se refieren a archivos que existen externamente al programa). Lo más probable es que los archivos reales se almacenen en un disco.

En BASIC, la correspondencia entre los archivos reales y los externos (los números de canal del BASIC) se define por medio de una sentencia como

```
FILE #3: ''DATOS DEL ESTUDIO''
```

En PASCAL no existe equivalente alguno a la sentencia FILE. En lugar de ello, hemos de especificar, en el encabezamiento del programa (es decir, en la construcción **program**, al principio mismo), todos los archivos externos que nuestro programa necesita. Esto explica por qué, para programas sencillos, escribimos como encabezamiento

program *x*(*input, output*);

Ello significa que los archivos *input* y *output* han de ser conocidos en el mundo exterior. Como mostramos en el capítulo 5, el encabezamiento de un programa es análogo en cierta forma al encabezamiento de un procedimiento. En el caso del encabezamiento del programa, todos los parámetros son nombres de archivos, y es por este medio como el programa interacciona con su entorno exterior.

Antes de explicar otros archivos externos, concretaremos algunas cosas acerca del uso de *input* y *output* en los encabezamientos de programas. Hay algunos programas que, en realidad, no tienen ninguna entrada o *input*, y otros no toman ninguna entrada o *input* del archivo *input* por defecto. En estos ca-

sos, se puede omitir el parámetro *input* en el encabezamiento del programa. Es difícil imaginar un programa que no utilice *output* o salida alguna. Incluso un programa para copiar un archivo en otro, el cual no utilizaría *output* para su salida normal, todavía podría necesitar el *output* para posibles mensajes de error. De hecho, algunos sistemas PASCAL insisten en que se mencione *output* en el encabezamiento del programa.

Correspondencia entre archivos externos y reales

Cuando tu programa emplee archivos externos distintos de los de *input* y *output*, éstos han de aparecer también en el encabezamiento. Por ejemplo,

program *análisis*(*input, output, f*1);

Para las reglas preferentes relativas a la ordenación de los nombres de estos archivos externos, consulta tu manual local; en ciertos sistemas, lo más conveniente sería poner *f*1 al principio de la lista. Algunos sistemas PASCAL permiten poner un asterisco a continuación de los archivos que son para sólo lectura; esto constituye una protección contra la escritura accidental en ellos. Dado el encabezamiento de programa que antecede, el nombre del archivo real que corresponde a *f*1 viene definido entonces por la orden o comando de tu sistema operativo que inicie una ejecución (*run*) en PASCAL. Por ejemplo, podría ser

RUN *análisis*(,,DATOS DEL ESTUDIO)

o

RUN *análisis* *f*1 = DATOS DEL ESTUDIO

(Algunos sistemas utilizan en lugar de "RUN" la palabra, más dramática, "EXECUTE".) Los sistemas operativos varían mucho, de manera que tendrás que consultar tu manual local. En la mayoría de los casos, lo que se empleará por defecto u omisión para *input* y *output* será tu terminal, pero puedes anular esta disposición si lo deseas. (En nuestros dos ejemplos de RUN, se han dejado sin especificar los archivos correspondientes a *input* y *output*. Esto significa que se aplicarán los correspondientes por defecto u omisión (*default*). Si en lugar de esto deseas que *input* y/o *output* vayan a otros sitios, puedes especificar los nombres de archivos reales.)

En muchos sistemas operativos, la regla en los casos de defecto u omisión para archivos distintos de *input* y *output* es que el nombre real sea igual que el del archivo externo. En el ejemplo precedente, y en ausencia de una especifi-

cación explícita, el nombre *f*1 serviría también como nombre real. Sin embargo, tales sistemas no son en modo alguno universales, y en general *no* se debe dar por supuesto que los nombres de los archivos externos se correspondan con los de los archivos reales (no más de lo que se corresponden con ellos los números de canal del BASIC).

Todos los archivos externos, distintos de *input* y *output,* que especifiques en el encabezamiento del programa deberán declararse también en la sección **var** del programa principal. Así, si el encabezamiento es

program *xxx*(*input, output, f*1, *f*2, *f*3);

serían necesarias las declaraciones de *f*1, *f*2 y *f*3, como las dadas anteriormente en este capítulo. En cambio, los archivos *input* y *output no han de ser* declarados; ya están declarados por defecto.

Dentro de tu programa PASCAL puedes crear algunos archivos temporales que no tengan existencia fuera del programa, o tal vez incluso fuera de un procedimiento del que el archivo sea local. Un ejemplo de esto podría surgir en un programa de ordenación o *sorting*, el cual crea un archivo intermedio que contiene una clase parcial de sus datos. No necesitas preocuparte acerca de nombres reales para estos archivos temporales.

El problema que presenta el sistema PASCAL de nombres reales de archivo es que el número de archivos reales está determinado de antemano por el número de archivos externos incluidos en el encabezamiento del programa. Así, no es posible escribir (por ejemplo) un programa de fusión o intercalación (*merging*) que lo haga con un número indeterminado o arbitrario de archivos. (Ni tampoco es posible escribir en PASCAL un intérprete de BASIC, si tu BASIC permite un número arbitrario de SAVE y OLD en diferentes archivos reales.) Dada la severidad de esta restricción, algunos compiladores PASCAL tienen ampliaciones especiales para eludir el problema. Muchos amplían las sentencias *reset* y *rewrite,* como describiremos más adelante.

Características de los archivos en PASCAL

En PASCAL, todos los archivos son secuenciales. No existe mecanismo alguno para tener archivos de acceso aleatorio (*random access*), a menos que el compilador disponga de ampliaciones especiales. Esto significa que cuando leas un archivo has de empezar por el principio y progresar, componente por componente, a lo largo de él. Cuando escribes un archivo, lo único que puedes hacer es agregar cosas al final del mismo; no puedes cambiar nada que ya esté escrito, como no sea empezando otra vez desde cero y escribiendo de nuevo todo el archivo.

Por tanto, asociado con todos los archivos hay un punto de exploración

o, en la terminología del PASCAL, una *ventana* que mira a un componente de un archivo (o al final de éste). Nos referimos a la ventana del archivo *f* como *f* ↑; su tipo de datos es el tipo de componentes del archivo, y se puede utilizar como una variable ordinaria. El valor de *f* ↑ es sinónimo del valor del componente actual.

Los procedimientos *get* y *put*

Si quieres realizar *inputs* y *outputs* con un nivel de detalle inferior al conseguido con *read* y *write,* el PASCAL provee cierto número de procedimientos incorporados que pueden ocuparse de los archivos al nivel de ventana. Los más importantes de ellos son el *get,* que se usa para lectura o entrada, y el *put,* que se usa para escritura o salida.

La llamada

```
get(f);
```

hace avanzar la ventana del archivo *f* para que mire o apunte al componente siguiente de dicho archivo.

El código o clave

```
var
    tfile: file of t; (* en donde t es cualquier tipo *)
    x: t; (* una variable del tipo t *)
begin
    x:= tfile↑;
    get(tfile);
```

lee en *x* el valor del componente actual, y hace que la ventana avance hasta después de éste. Esto es precisamente lo que hace el procedimiento *read* cuando se le llama con *x* como argumento. Por tanto, el código precedente equivale a

```
read(tfile, x);
```

Normalmente, todos los *inputs* se realizarán usando *read,* sin preocuparse del *get,* pero a veces es conveniente programar a un nivel más bajo y utilizar las ventanas y el *get*. Las mismas, o análogas, consideraciones, son de aplicación al *write* y a su procedimiento correspondiente de bajo nivel *put*. (Algunos compiladores siguen al informe del PASCAL en cuanto a restringir el uso de *read* y *write* a los archivos de texto o *textfiles,* aunque la norma PASCAL los permite para los archivos binarios. Si tienes un compilador tan restringido, ne-

cesitarás rehacer los prohibidos *read* y *write* al nivel inferior; una tarea ligeramente cansada, pero que no es onerosa. Esta tarea habría de hacerse, por ejemplo, en el programa sumador de archivos que mostramos más adelante en este capítulo.)

Obsérvese que, por lo que respecta a los archivos de texto, o *textfiles,* un *read* puede hacer más que un *get.* Si leemos (*read*) un número real, por ejemplo, el PASCAL toma (*gets*) automáticamente todos los caracteres que forman el número, y los convierte a la forma binaria interna.

Hay un procedimiento de salida (el *put*) que es el correspondiente al de entrda *get.* La llamada

```
put(f);
```

añade la ventana *f* ↑ al final del archivo *f.* Antes de hacer el *put* se debe asignar algo a la ventana. El archivo *f* ha de estar posicionado en el final de archivo antes de hacer el *put.* Esto es otra forma de decir que los archivos han de escribirse secuencialmente, que sólo se puede escribir en ellos agregando lo que sea al final.

La sentencia

```
write(tfile, x);
```

es equivalente a

```
tfile↑:= x;
put(tfile);
```

(Otra vez, sin embargo, los archivos de texto pueden ser una excepción. La escritura, por ejemplo, de un número real en un archivo de texto es más complicada que un simple *put,* ya que implica una conversión y la salida *(output)* de varios caracteres.)

El siguiente fragmento de programa ilustra el empleo de *get* y *put.* Lee el archivo *input* hasta encontrar la primera vez que se produce el carácter ":"; todo el texto sobre el cual se pasa, se copia al archivo *output.*

```
while input↑ < > ':' do
begin
    output↑:= input↑;
    put(output);
    get(input);
end;
```

Tal como está, este programa tiene dos defectos, que explicaremos más adelante. Por tanto, nos referiremos de nuevo a este programa buscador del signo "dos puntos".

Si deseas hacer algo "cuestionable", como leer un archivo usando *get* y luego escribir al final del mismo archivo usando *put,* consulta el manual de tu PASCAL local. Lo que puedas y no puedas hacer viene dictado más veces por el sistema operativo local que por las reglas propias del PASCAL.

Inicialización de archivos

El PASCAL provee un procedimiento incorporado

rewrite(f);

que prepara un archivo para la escritura; lo hace destruyendo el contenido anterior del archivo (si lo tenía), y situando la ventana en el principio de aquél. En este caso, el principio del archivo es también su final, porque el archivo es nulo.

Existe un procedimiento similar

reset(f);

que prepara un archivo para la lectura. Este sitúa la ventana en el principio del archivo, lista para la lectura. En este caso, el principio del archivo no es normalmente el final, o no habría nada que leer.

Antes de usar cualquier archivo que hayas declarado, deberás hacer un *reset* o *rewrite* del mismo, según corresponda.

Sin embargo, los archivos incorporados *input* y *output* se preparan automáticamente para su lectura y su escritura (respectivamente), y nunca se deberán inicializar.

> "Todo muy lógico", dijo Bill con cierto sarcasmo en la voz. "No entiendo por qué hacéis un *rewrite* a fin de prepararos para escribir, pero no un *reread* cuando os preparáis para leer. El mismo nombre *rewrite* me parece también un poquitín extraño: "re-escribís" antes de "escribir". Indudablemente, si yo fuera tan inteligente como el profesor Primple, entendería estas cosas."

La función *eof*

Frecuentemente se desea leer datos de entrada hasta que se terminan, es decir, hasta que se llega al final del archivo de entrada. En BASIC se puede

detectar el final de un archivo por medio de una sentencia tal como IF END. El PASCAL provee una función Booleana incorporada

eof(*f*)

que es verdadera (*true*) sólo si el archivo *f* está posicionado al fin de un archivo. Por consiguiente, sólo se puede escribir (*put* o *write*) en un archivo si *eof* es verdadero (*true*), y sólo se puede leer de él (*get* o *read*) si *eof* es falso (*false*).

Un defecto que tiene nuestro programa buscador de ''dos puntos'' es que si no encontrara ningún signo de este tipo seguiría leyendo hasta pasarse del final del archivo. Sería mejor usar la función *eof* para comprobar esta condición de error y, si surgiera, producir un mensaje de error.

El programa completo que sigue ilustra el uso de los archivos y el *eof*. El programa suma una serie de números reales que hay en el archivo de entrada; cuando se llega al final del archivo, el programa imprime la suma total y se para.

```
program sumarchivo(datafile, output);
(* Programa para sumar todos los componentes de un archivo *)
var
    datafile: file of real;
    sum: real;
    x: real;
begin
    reset(datafile);
    sum:= 0;
    while not eof(datafile) do
    begin
        read(datafile, x);
        sum:= sum + x;
    end;
    writeln('Sum = ', sum);
end.
```

Este ejemplo ilustra varios puntos que hemos hecho notar antes. Muestra cómo un archivo externo, tal como *datafile,* que aparece en el encabezamiento del programa, tiene que ser declarado entre todas las demás variables al comienzo del mismo. El archivo *output,* en cambio, no se declara.

Este programa no usa el archivo *input,* que, por consiguiente, se ha omitido en el encabezamiento. Sin embargo, no habría importado que se hubiera incluido redundantemente en él. Muchas personas escriben siempre

```
program xxx(input, output);
```

como primera línea en todos sus programas (siempre que no utilicen archivos externos), independientemente de si en realidad se usa *input*.

El archivo *datafile* se declara como un archivo de números *reales*. El pro-

grama podría ajustarse para trabajar sobre un archivo de texto *(textfile)* en lugar de uno binario, declarando el archivo de datos como

```
datafile: text;
```

Existen, sin embargo, problemas en el uso de *eof* en archivos de texto. Más adelante, en este mismo capítulo, definiremos una función *texteof,* que deberá usarse en lugar de la *eof* del programa precedente si su entrada es desde un archivo de texto.

Por lo que respecta a la forma de funcionamiento del programa, debe ser fácil seguirla. Simplemente, lee los datos, acumulando la suma, hasta que se llega al final del archivo. Obsérvese que el programa funciona incluso si *datafile* es nulo.

Los archivos como parámetros

Es instructivo escribir de nuevo el programa precedente en forma de procedimiento que tome como parámetro un archivo y escriba la suma de los componentes del mismo.

Supondremos que deseamos sumar tres archivos separados, de modo que el uso de un procedimiento se hace natural. El programa para realizarlo es el siguiente:

```
program suma3archivos(archivo1, archivo2, archivo3, output);
(* Suma los componentes de cada uno de los tres archivos *)
type
    realfile = file of real;
var
    archivo1,archivo2,archivo3: realfile;
procedure sumarchivo(var datafile: realfile);
(* Imprime la suma de todos los componentes de datafile *)
var
    sum: real;
    x: real;
begin
    reset(datafile);
    sum:= 0;
    while not eof(datafile) do
    begin
        read(datafile, x);
        sum:= sum + x;
    end;
    writeln('Sum= ', sum);
    end; (* sumfile *)
```

```
begin (* programa principal *)
    sumfile(archivo1);
    sumfile(archivo2);
    sumfile(archivo3);
end.
```

Obsérvese cómo el parámetro para *sumfile* se ha declarado como un parámetro **var**. Si esto se omitiera, la primera acción al llegar a

sumfile(*archivo*1);

sería copiar el argumento en el parámetro. Esto exigiría copiar todo el archivo1, lo cual es algo tan desaforado que el PASCAL lo prohíbe absolutamente. Los parámetros de archivos han de ser siempre parámetros **var.**

El programa sirve también para ilustrar un problema suscitado antes. Se trata de que todos los archivos externos han de declararse en el encabezamiento del programa y que, por consiguiente, no hay forma en PASCAL de crear dinámicamente un nombre de archivo. Ello significa que no existe equivalente de la sentencia de BASIC

```
PRINT "TECLEE EL NOMBRE DEL ARCHIVO QUE SE VA A
      USAR:";
INPUT S$
FILE #3: S$
```

(aunque, como ya hemos dicho, tu PASCAL local puede tener una ampliación —tal vez un parámetro adicional para *reset* y *rewrite*— que ayude a conseguir el mismo efecto). Por tanto, si deseáramos sumar cuatro archivos en vez de tres, tendríamos que cambiar el programa anterior en tres formas:

- para añadir el *archivo*4 al encabezamiento del programa;
- para declarar el *archivo*4;
- para aplicar *sumfile* al *archivo*4.

Operaciones sobre archivos

"¿Qué pasaría si yo quisiera aplicar *sumfile* a *input*?", dijo Bill, con un brillo en sus ojos que desmentía la aparente inocencia de su pregunta.

Desgraciadamente, la respuesta es que el programa fallaría, porque *sumfile* sitúa a su parámetro en el estado inicial (*reset*), y no se puede hacer eso con *input*. Ni tampoco se puede incluir dentro del procedimiento una comprobación tal como

if *datafile* < > *input* **then** *reset*(*datafile*);

No es posible comparar nombres de archivos.

Esto recuerda en cierto modo a los registros. No se puede hacer nada con los registros excepto asignarlos. Los archivos, sin embargo, son todavía un poco peores. No es posible ni siquiera asignar un archivo a otro. Todo lo que se puede hacer es pasarlos como parámetros **var,** usarlos como argumentos para procedimientos incorporados, tales como el *get,* y emplearlos como referencia para la venta actual.

Los *cokneys* londinenses, que pronuncian *fail* (fracasar) lo mismo que *file* (archivo), es posible que al hacerlo estén pensando en el PASCAL.

Archivos y matrices

Ya hemos observado que la declaración de un archivo se parece en algo a la de una matriz o *array,* como evidencian las declaraciones

a: **array** [1..100] **of** *real*;
f: **file of** *real*;

Si mencionamos las diferencias entre estos dos conceptos, ello reforzará algunas de las propiedades de los archivos del PASCAL.

En primer lugar, puede verse que un archivo, a diferencia de un *array* o matriz, puede tener un tamaño ilimitado, sujeto, desde luego, al de tu máquina y al de su memoria de reserva o apoyo. Ello no significa, sin embargo, que, si declaras de nuevo a todas tus matrices como archivos, vayan a desaparecer todos tus problemas. Los problemas surgen porque sólo podemos leer los archivos secuencialmente y (lo que es peor) sólo podemos escribirlos secuencialmente. Lo cual quiere decir que, incluso si cambiamos un solo elemento, tenemos que escribir de nuevo todo el archivo.

En segundo lugar, si alternamos la lectura y la escritura, tenemos que estar continuamente haciendo *reset* y *rewrite* en el archivo.

Así, con sólo unos cuantos programas, puedes utilizar los archivos en forma cómoda y conveniente para almacenar grandes matrices o *arrays.* Sin embargo, si tu PASCAL local te ofrece archivos de acceso aleatorio, y si no te preocupa excesivamente la portabilidad, las oportunidades serán mucho mayores.

Propiedades especiales de los archivos de texto

Hasta ahora, la mayor parte de todo lo que hemos dicho es aplicable por igual a los archivos de texto y a los binarios. La única facilidad adicional que

hemos encontrado para los archivos de texto ha sido la capacidad para leer y escribir diversos tipos de datos diferentes y para convertirlos a forma interna.

Los archivos de texto poseen además otra importante propiedad: están divididos en líneas, las cuales van separadas por caracteres de "cambio de línea", o "retorno de carro". En algunos lenguajes, estos cambios de línea son caracteres reales, que se pueden manipular al igual que cualquier otro. En otros, como el BASIC, el cambio de línea es un carácter hipotético. El PASCAL adopta una posición intermedia: el cambio de línea cuenta como un carácter, pero cuando lo leemos obtenemos un espacio. Así, si los datos de entrada del archivo de texto (*input*) constan de las líneas

```
23
4
x...
```

y ejecutamos el programa

```
for k:=1 to 5 do
begin
    write(input↑);
    get(input);
end;
```

con la ventana posicionada inicialmente en el carácter "2", nuestra salida estará formada por los cinco caracteres "2", "3", espacio, "4", espacio. La ventana termina apuntada al carácter "*x*".

Esto pone de manifiesto el segundo defecto que tiene nuestro programa buscador del carácter "dos puntos": a medida que va copiando la entrada a la salida, todos los cambios de línea se convierten en espacios.

Aunque el cambio de línea nos aparezca como un espacio, hay un medio para detectar su presencia. Esto se hace por medio de la función incorporada *eoln,* que es similar a la función *eof.* La función *eoln* sólo devuelve o produce el valor *true* si la ventana está posicionada en el fin de una línea.

La función *eoln* se puede utilizar para eludir una de las limitaciones del PASCAL. El fragmento de programa que mostramos a continuación lee una línea de caracteres y los reúne formando con ellos una cadena o *string.* En esta forma, consigue el efecto de la lectura de una cadena. El código, que al mismo tiempo introduce una cadena y la imprime en la salida, es el siguiente:

```
const
    maxlonglínea = 80;
var
    linentrada: packed array [1..maxlonglínea] of char;
    k: integer;
    longlínea: 0..maxlonglínea;
```

```
begin
    for k: = 1 to maxlonglínea do (* borrar la línea *)
        linentrada[k]: = '';
    longlínea: =  0;
    while not eoln(input) do
    begin
        if longlínea < maxlonglínea then
           longlínea: = longlínea  +  1
        else
           (* ... dar mensaje de error... *);
        read(linentrada[longlínea]);
    end);
(* Ahora pasa el carácter de cambio de línea, de modo que la próxima
   entrada llegue en una línea nueva *)
        get(input); (* o, alternativamente (véase más adelante): readln; *)

(* Ahora escribe la línea que acaba de leer *)
        writeln(linentrada);

(* Lo que precede escribe todos los espacios que haya al final de la
   linentrada. Las sentencias que siguen constituyen un perfeccionamiento
   que elude esos espacios
    for k: = 1 to longlínea do
        write(linentrada[k]);
    writeln;
*)
```

Los procedimientos *writeln* y *readln*

En BASIC existe una gran diferencia entre

```
    PRINT X;
y   PRINT X
```

Como sabes, la última de estas sentencias pone fin a la línea actual (da salida a un cambio de línea, si lo prefieres), mientras que la primera permite que se produzca más salida en la misma línea.

En PASCAL, como ya hemos indicado en el capítulo 6, los equivalentes de las sentencias BASIC anteriores son

```
    write(x);
y   writeln(x);
```

así, el procedimiento *writeln* pone fin a una línea, mientras que el *write* no lo hace. Del mismo modo exactamente que en BASIC se puede emplear

PRINT

como una sentencia por sí sola, en PASCAL se puede utilizar

writeln;

por sí sola. Una sucesión de sentencias tales como

writeln; *writeln*; *writeln*;

es útil para escribir una serie de líneas en blanco. Al igual que *write, writeln* tiene como primer argumento un nombre opcional de archivo. Suponiendo que *f* sea un nombre de archivo, lo que sigue son ejemplos de lo que hemos dicho:

writeln(*f, x*);
writeln(*f*); *writeln*(*f*); (* *dos líneas en blanco en el archivo f* *)

El PASCAL soporta también un procedimiento incorporado *readln,* que no tiene equivalente directo en BASIC. Del mismo modo que *writeln* añade un cambio de línea una vez que haya salido el último de sus argumentos, haciendo así que la salida posterior aparezca en una nueva línea, *readln,* una vez que se ha leído el último de sus argumentos, salta el resto de la línea actual, de modo que los datos posteriores se tomen de la línea siguiente. Así, si *k* es una variable entera (*integer*), y la sentencia

readln(*k*);

se ejecuta con los datos de entrada

23 24 *se ha perdido*
x ...

k tomará el valor 23, se saltarán los datos "24 se ha perdido", y la ventana se quedará apuntando al carácter *x* del comienzo de la nueva línea.

Obsérvese que en una sentencia como

readln(*a, b, c*);

es completamente legal que los datos correspondientes a *a, b,* y *c* estén bien en una sola línea o bien en tres líneas separadas, como es normal para un *read.* La única diferencia que existe en relación con un *readln* es que salta hasta después del primer cambio de línea siguiente al valor suministrado para *c.*

Al igual que *writeln, readln* se puede emplear aislada o sola, sin lista de argumentos, para saltar al comienzo de una línea nueva.

El PASCAL proporciona también, como adjunto o complemento de *writeln,* otro procedimiento incorporado

page(*f*)

que inicia una página nueva en el archivo *f*.

Estos procedimientos, *readln, writeln* y *page,* sólo se pueden utilizar en archivos de texto.

El final-de-archivo en los archivos de texto

En muchos sistemas operativos, el final de un archivo de texto sólo puede venir después de una línea completa. Es como si el final-de-archivo fuese un carácter imaginario que viniese al comienzo de una línea. Esto repercute sobre el uso de la función *eof* en archivos de texto. Cuando se compruebe si ha llegado el *eof,* deberemos tener posicionada la ventana en el principio potencial de una línea. Por consiguiente, las siguientes líneas de programa no funcionarán:

```
read(x);
if eof(input) then (* ... *);
```

La razón de ello es que *read* deja la ventana en el carácter siguiente al número que se acaba de leer. Suponiendo que el usuario suministrase la línea

23

la ventana estaría apuntada al carácter de cambio de línea posterior al ''3''. (Recuerda que, si miras efectivamente a este carácter, éste se convierte en un espacio.) Por tanto, *eof* será siempre *false.*

Para solucionarlo, se deberá leer el carácter de cambio de línea, de modo que la ventana quede apuntada al comienzo de la línea siguiente, cosa que se puede hacer por medio de

```
read(x, nextchar);
```

en donde *nextchar* se declara como una variable *char,* o, mejor todavía, por medio de

```
readln(x);
```

Esto último es un perfeccionamiento, ya que absorbe cualesquiera espacios (o, en realidad, cualesquiera otros caracteres) que sigan al valor de *x,* pero sólo es aplicable si se sabe que el elemento de datos suministrado para *x* se

encuentra al final de una línea. (De no ser así, los "otros caracteres" que nos saltáramos podrían corresponder a datos vitales.)

Hay, no obstante, una solución más limpia y más fiable que cualquiera de estas dos alternativas. Consiste en escribir una función *texteof,* que equivale a "saltar los espacios (si los hay) hasta que se llegue a un carácter que no sea espacio, o al final de una línea, y entonces realizar *eof".* Una vez que se haya definido esta función, puedes usarla libremente en lugar de *eof* siempre que estés explorando un archivo de texto en contextos en los que los espacios no tengan significado. Este es un buen ejemplo de la creación de una herramienta para la realización de lo que, en otro caso, sería una tarea difícil.

La función *texteof* se define en la forma siguiente:

```
function texteof(var f: text): Boolean;
(* Explora a lo largo del archivo f hasta llegar a un no-espacio o al final
   de una línea; produce el valor true si f está entonces en eof *)
var
   paraexplor: Boolean; (* false si queremos seguir adelante exploran-
                           do en busca de espacios *)
begin
   repeat (* bucle para explorar los espacios que haya *)
      paraexplor:= true; (* el valor de defecto, o default *)
      if not eof(f) then
         if f↑ = ' ' then
         begin
            paraexplor:= eoln(f); (* no explorar más allá del final de
                                     línea *)
            get(f);
         end;
   until paraexplor;
   texteof:= eof(f);
end; (* texteof *)
```

Entrada/salida interactivas

Los primeros implementadores del PASCAL visualizaron los dispositivos de entrada/salida como lectores de tarjetas, impresoras de línea y otros semejantes.

Cualquier intento de interacción utilizando una terminal se enfrentaba al desastre. Incluso el intento de preparar un programa PASCAL en cualquier otro medio que no fuese las tarjetas estaba erizado de riesgos, tales como el que se tomase como un número de secuencia todo aquello que apareciese después de la columna 72. Algunos de estos compiladores andan todavía por ahí, induciendo a sus usuarios a entregarse a la bebida o al BASIC.

Vamos a mencionar algunos de los problemas, a fin de que puedas reír ahora, antes de las lágrimas que vendrán después.

Uno de ellos es que algunos compiladores PASCAL conservan la información de salida en *buffers* (o memorias temporales) más bien extensos, de, por ejemplo, 1.024 caracteres. La información sale de ellos sólo cuando el *buffer* está lleno o cuando el programa haya terminado de ejecutarse. Entonces, la conversación con los programas de PASCAL adopta un aire bastante curioso. Uno puede insistir en sus preguntas o comunicaciones en la entrada, y el ordenador permanece callado. Luego, de repente, cobra vida, disparándonos un chorro de 1.024 caracteres y respondiéndonos a preguntas que habíamos tecleado hacía un buen rato. (Estos *buffers* pueden ocasionar problemas cuando hay fallos de programa a causa de errores; a menudo se pierde el último *buffer* incompleto, con el resultado de que la depuración se convierte en un bonito juego de las adivinanzas.) Los *buffers,* desde luego, están pensados para hacer más eficientes las transferencias a y desde cinta magnética y disco, y el fallo interactivo es un desafortunado efecto secundario.

Los *buffers* se aplican a las entradas, al igual que a las salidas. Concretamente, muchos sistemas PASCAL procesan la entrada interactiva línea a línea, con el resultado de que no examinan la entrada hasta que se pulsa la tecla RETURN. Otros sistemas PASCAL, principalmente en microordenadores, ofrecen la entrada en el llamado ''modo no elaborado'' *(raw mode),* mediante el cual se trata o procesa inmediatamente cualquier carácter que teclee un usuario interactivo. Consulta tu manual local en busca de estos detalles; es muy posible, en cualquier caso, que te encuentres con que la entrada interactiva es objeto de un tratamiento no-estándar.

Otro problema que surge está relacionado con la definición de *read.* El PASCAL hace avanzar la ventana hasta el carácter siguiente al último que se haya leído. Si este último se encuentra al final de una línea —o si ya se ha empleado *readln* en vez de *read*—, el PASCAL hace avanzar la ventana hasta el inicio de la siguiente línea. Muchos compiladores de PASCAL nos piden en este momento que introduzcamos la línea siguiente. Esto se hace antes de haberse tratado la línea tecleada anteriormente. El resultado, en condiciones estables, es que siempre estamos tecleando con un adelanto de una línea sobre la que en cada momento acabe de ser tratada. En consecuencia, si el programa produce invitaciones o avisos *(prompts),* éstos se referirán a la línea que se había tecleado anteriormente. Así, una conversación podría desarrollarse en la forma siguiente (siendo lo escrito en minúsculas la parte correspondiente a la máquina):

> ...JUAN
> *¿Cómo te llamas*? 26
> *¿Cuántos años tienes*? NO EL PASCAL
> *¿Cuál es el mejor lenguaje de programación en cuanto a entrada/salida*?

Las cosas se hacen aún peores si nos fallan los poderes psíquicos y alguna de las respuestas nos sale equivocada. En conjunto, el resultado es que uno

de los mayores placeres de la vida, como es la conversación, se queda bastante aguado.

También se producen problemas parecidos si nuestro programa imprime una respuesta para cada línea de entrada, por ejemplo:

.
.
.
64
no es una potencia de dos
23
es una potencia de dos
32
no es una potencia de dos

(Cada respuesta se refiere a la penúltima entrada.)

Llamaremos a éste el problema del *desfasamiento*. Si tu compilador PASCAL presenta este tipo de problemas, tal vez puedas escribir algunos procedimientos un tanto arfificiosos que te ayuden a vencer las dificultades. Véase en Kaye (1980) un buen estudio de los mecanismos implicados en ello. Otra alternativa es pasarse a un compilador que haya pensado algo en el usuario interactivo.

Un último problema relacionado con la entrada interactiva es que el PASCAL da por sentado que la persona situada en la terminal es perfecta y no se equivoca nunca. Si los datos tienen un formato equivocado —por ejemplo, si se introduce una letra allí donde se espera un número—, el PASCAL da un mensaje de error y pone fin inmediatamente a la ejecución.

Una conversación, en un programa de enseñanza con ayuda del ordenador, podría desarrollarse en la forma siguiente (suponemos aquí que el problema del desfasamiento está ya resuelto):

¿Cuántas son dos y dos?
5
Lo siento, Jaime, eso no está del todo bien, prueba de nuevo.
$ (*a Jaime se le ha olvidado pisar la tecla de las mayúsculas para el* '4')
Error en los datos de entrada
Procedimiento lecturarespuesta interrumpido
Fin de la ejecución.

Entonces, es probable que las teclas de la terminal se humedezcan con las lágrimas de Jaime.

No existe forma alguna de eludir este problema (a menos que tu PASCAL tenga una ampliación o extensión no estándar), como no sea escribir tus procedimientos propios que tomen la entrada carácter por carácter, y la conviertan a la forma numérica cuando ello sea necesario. Entonces, si existe cualquier error, podrás dar al usuario un mensaje amistoso y pedirle que lo intente de nuevo.

Comprobación interactiva del fin-de-archivo, o *eof*

Incluso si tienes un buen compilador interactivo, habrás de ser cuidadoso si quieres vencer completamente el problema del desfasamiento. El código que sigue pone de manifiesto el problema:

```
suma:= 0;
while not eof(input) do
begin
    write ('Suma hasta ahora = ', suma, '; teclee el número siguiente:');
    readln(x);
    suma:= suma + x;
end;
writeln('Suma total = ', suma);
```

En este programa se utiliza *readln,* como recomendamos anteriormente, porque entonces el *eof* funciona correctamente. (Es posible que hubiera sido mejor emplear la función *texteof,* pero ello habría hecho este comentario más complicado.)

Aun en el caso de que tu compilador interactivo sea lo bastante inteligente para no desfasarse a causa del mismo *readln,* todavía lo hará como consecuencia de un problema lógico asociado con la función *eof.* La única forma en que el PASCAL puede saber si hay un final de archivo, es pedirte otra línea de entrada y ver si la hay o no. (Suponemos que el fin de archivo en una terminal se indica tecleando algún carácter especial al comienzo de la línea.) Así, al usuario se le pide que teclee una línea todas las veces que se ejecute **while not** *eof*(*input*). Esta línea es utilizada entonces por la sentencia *readln* que viene tres líneas más abajo. El efecto es que se pide al usuario que suministre su línea de entrada unas sentencias antes de lo que un vistazo poco atento al programa podría indicar. El problema es que hay un *write* que viene después del *eof,* pero antes que el *readln.* Este *write,* por tanto, se ejecuta después de haber dado entrada a una línea, pero antes de que la misma haya sido tratada o procesada. El resultado familiar es que la entrada se desfasa, por ejemplo:

```
23
Suma hasta ahora = 0; teclee el número siguiente: 46
Suma hasta ahora = 23; teclee el número siguiente: (FIN DE ARCHI-
                                                    VO, o eof)
Suma total = 69
```

Este es un problema lógico, no de tu compilador PASCAL, y la única forma de eludirlo es escribir de nuevo el programa de modo que el *write* venga antes que el *eof.* En el caso precedente, el resultado, un tanto lioso, es:

```
suma:= 0;
write('Suma hasta ahora= 0; teclee el número siguiente:');
while not eof(input) do
begin
    readln(x);
    suma:= suma + x;
    write('Suma hasta ahora =', suma,'; teclee el número siguiente:');
end;
writeln('Suma total= ', suma);
```

Dado que la comprobación interactiva de *eof* te va a ocasionar problemas, el mejor consejo es que lo pienses cuidadosamente antes de utilizarla. ¿Sería mejor formular tu programa de otra manera que permita evitar el *eof*? Una alternativa es preguntar una y otra vez al usuario interactivo si tiene más datos, y parar cuando conteste negativamente. Otra es emplear algún valor determinado que funcione como terminador para el conjunto de datos de que se trate; frecuentemente, el valor cero es adecuado para ello.

Esta misma extremada cautela ha de aplicarse al uso interactivo de *eoln*.

Formatos de salida

Como último tema importante de este capítulo, consideraremos ahora los *formatos de salida*. Al llegar a este punto, hemos de confesar que los *writes* de todos los ejemplos dados hasta ahora en este libro han empleado unos formatos de salida muy poco refinados, y que en muchos casos sus resultados no serían agradables de ver; el PASCAL provee un medio para lograr algo mejor.

Hay facilidades análogas, pero más potentes, en aquellos BASIC que permiten el empleo de una cláusula USING en una sentencia PRINT. La cláusula USING se aplica a la sentencia PRINT entera, pero las facilidades de control de formato del PASCAL se aplican a argumentos individuales de *write* o *writeln*. La más útil de las facilidades PASCAL es la capacidad para agregar, a un elemento que se haya de imprimir, un número entero (*integer*) que represente una *anchura mínima de campo*. El ejemplo que sigue ilustra la sintaxis correspondiente:

```
wrinteln('Los totales para case', caseno: 2, 'son', sumaa: 10, sumab);
```

Como puede verse, se escribe un signo de ''dos puntos''(:) después de cada elemento que haya de tener una anchura mínima de campo. En este ejemplo, *caseno* (que suponemos que es un entero o *integer*) tiene una anchura mínima de campo de dos; esto significa que normalmente se imprimiría con dos caracteres. La regla es bastante directa si *caseno* está entre 10 y 99. Si *caseno* está formado por una sola cifra, la regla es que se alinee por la derecha, es decir,

poniendo un espacio delante. Si *caseno* es mayor de 99, es evidente que no cabrá en dos caracteres. Sin embargo, normalmente esto no constituye un error en PASCAL; lo que se hace es imprimir el valor utilizando el menor número de caracteres posible. Recuérdese que el entero 2 representa la anchura *mínima* del campo, y no necesariamente la utilizada en realidad. Así pues, cuando queramos que un elemento se imprima siempre con el menor número de caracteres posible, podemos darle 1 como anchura mínima de campo.

Las anchuras mínimas de campo se pueden aplicar a elementos de cualquier tipo de datos, aunque su aplicación más frecuente es con enteros y reales. Si se omite la anchura mínima de campo —como hemos venido haciendo en todos nuestros ejemplos hasta que hemos llegado a esta sección—, tendremos una anchura de campo que vendrá definida por la implementación para el tipo de datos de que se trate. Consulta los detalles en tus manuales PASCAL locales. En nuestro ejemplo precedente, *sumaa* se imprime con una anchura de 10, pero *sumab* se imprime con la anchura de campo que tenga asignada por defecto su tipo de datos.

En la práctica, estas anchuras asignadas por defecto suelen ser largas, con lo que, si hubiéramos omitido el ":2" después de *caseno,* una línea de salida podría tener este aspecto

los totales para el case   4 *son* ...

Parece que el Sr. 869704 ha persuadido a los escritores de compiladores de que la salida usualmente está formada por columnas muy separadas de cifras, en lugar de por una combinación legible de texto y números.

Longitud de la parte fraccionaria

Si se especifica una anchura mínima de campo, pueden ir tras ella otros dos puntos y otro entero, al que se denomina la *longitud de la parte fraccionaria,* por ejemplo:

writeln(*'Suma= ', sumaa*: 8: 3, *' unidades.'*);

Si se expresa una longitud para la parte fraccionaria, el valor a imprimr ha de ser un número real. En el ejemplo precedente, la variable *sumaa* se imprime con una anchura mínima de campo de 8 y una longitud de la parte fraccionaria de 3, lo cual significa que habrá tres cifras después del punto decimal. (*N. del E.:* Debido a que en la notación del ordenador de los números decimales se emplea el punto en lugar de la coma del castellano, hemos referenciado con "punto flotante" en lugar de "coma flotante" a lo largo del texto. De igual forma hemos procedido en cuanto a "punto decimal" en lugar de "coma decimal".) Así, si *sumaa* tiene el valor 12.34567, la salida sería

Suma= 12.346 *unidades.*

Obsérvese que se imprimen dos espacios delante del valor de *sumaa,* asegurando así que la anchura del campo alcance el mínimo de 8.

De hecho, la presencia de la longitud de la parte fraccionaria hace que cambie la forma en que se imprimen los números reales. Si no se expresa longitud alguna para la parte fraccionaria, el valor sale en notación de "punto flotante", por ejemplo:

$$1.2345670000E+01$$

Por tanto, la longitud de la parte fraccionaria es sin duda un "bueno", porque hace que los números salgan en la forma que, para casi todos nosotros, es más legible.

Resumiendo, los valores por defecto que obtenemos si omitimos la anchura mínima de campo o la longitud de la parte fraccionaria, suelen ser lo que en realidad nos interesaba menos.

Otros puntos

Obviamente, las facilidades para controlar el formato sólo son aplicables si la salida va a un archivo de texto, y no a uno binario.

En realidad, las facilidades para el control de formato del PASCAL poseen una ventaja sobre la cláusula USING del BASIC. Esta ventaja consiste en que la anchura mínima de campo y la longitud de la parte fraccionaria pueden ser expresiones cuyos valores se pueden cambiar en el curso de la ejecución. Como ejemplo un tanto absurdo, la sentencia

```
for k:= 1 to 5 do
    writeln(k: k);
```

produciría esta salida:

```
1
 2
  3
   4
    5
```

De un modo más práctico, estos controles de formato se pueden definir utilizando constantes con nombre, lo que hace más fácil cambiarlas, por ejemplo:

```
const
    anchuradecase = 2;
```

```
var
    caseno: integer;
begin
(*  .
    .
    .  *)
    writeln('Los totales para el case', caseno: anchuradecase, ' son...');
```

Como último punto concerniente a los formatos, obsérvese que la sentencia PASCAL

```
write(a, b, c);
```

se asemeja a la de BASIC

```
PRINT A; B; C;
```

más que a

```
PRINT A, B, C,
```

En otras palabras, el PASCAL no inserta espacios adicionales entre los elementos que hayan de imprimirse. Esto significa que usualmente somos nosotros los que tenemos que poner espacios en el comienzo y/o el final de las cadenas o *strings* en las que haya intercalados números. Por ejemplo,

```
writeln('Resultado del case ', caseno: 2);
```

produciría, si *caseno* tuviera el valor 18, la siguiente línea:

```
Resultado del case18
```

En consecuencia, la sentencia precedente estaría mejor expresada así:

```
writeln('Resultado del case ', caseno: 2');
```

Si se imprimen números positivos sin control de formato, se imprime un espacio para el signo (mientras que delante de los números negativos se imprime un signo menos); así pues, no hace falta incluir un espacio más al final de una cadena precedente.

Tipos de datos encarcelados o recluidos

Hemos indicado anteriormente cuáles eran los tipos de datos que podían leerse y escribirse en archivos de texto. En esta última sección diremos algo

más acerca de los infelices tipos de datos que están recluidos o encarcelados dentro del PASCAL, y que no pueden ser objeto de entrada y salida.

Los que más destacan entre estos prisioneros son los tipos definidos por el usuario. Si vuelves al programa *adelgazador* del capítulo 7, podrás recordar que había en él un tipo *día* que representaba los días de la semana. También contenía una variable *hoy* del tipo *día*. El programa podría mejorarse dando salida al valor de *hoy* en la tabla de resultados. Sin embargo, la sentencia

```
write(hoy);
```

no imprimiría el valor de *hoy,* que, por ejemplo, podría ser *martes;* en lugar de hacerlo, el compilador daría un mensaje de error. Si realmente quieres producir la salida de un objeto del tipo *día,* la mejor forma de hacerlo es escribiendo un procedimiento, que podría tomar la forma siguiente:

```
procedure escribedía(d: día);
(* Escribe el día de la semana representado por d *)
begin
    case d of
        lunes:
            write('Lunes');
        martes:
            write('Martes'):
        miércoles:
            write('Miercoles');
        jueves:
            write('Jueves');
        viernes:
            write('Viernes');
        sábado:
            write('Sábado');
        domingo:
            write('Domingo');
    end;
end; (* escribedía *)
```

La escritura de estos procedimientos es aburridísima. Su único mérito es que si tienes un amigo o un pariente de esos que siempre están ansiosos por ayudar, pero que no son muy buenos programando, estos procedimientos serán un pasto ideal para él, y a ti te permitirá conseguir un poco de paz.

Un procedimiento para entrada semejante a éste sería todavía peor, ya que los nombres de los días se han de leer y ensamblar carácter por carácter. Es poco probable que la tía Amy o el pequeño Willy consigan introducir correctamente el procedimiento.

Desde luego, es una verdadera lástima que los tipos definidos por el usuario sean prisioneros, ya que si no lo fueran resultarían aún más útiles.

También los registros se encuentran entre los prisioneros, aunque si se les trata campo por campo pueden escurrirse para entrar o salir. Este mal trato dado a los registros resultará una gran sorpresa para los lectores familiarizados con lenguajes tales como el COBOL, en los que uno de los fines centrales de los registros es definir los formatos de entrada/salida.

Desde luego, se puede introducir o extraer cualquier tipo de datos si se usa un archivo binario de dicho tipo; pero tales archivos no son legibles para los mortales, sino únicamente para los programas de ordenador.

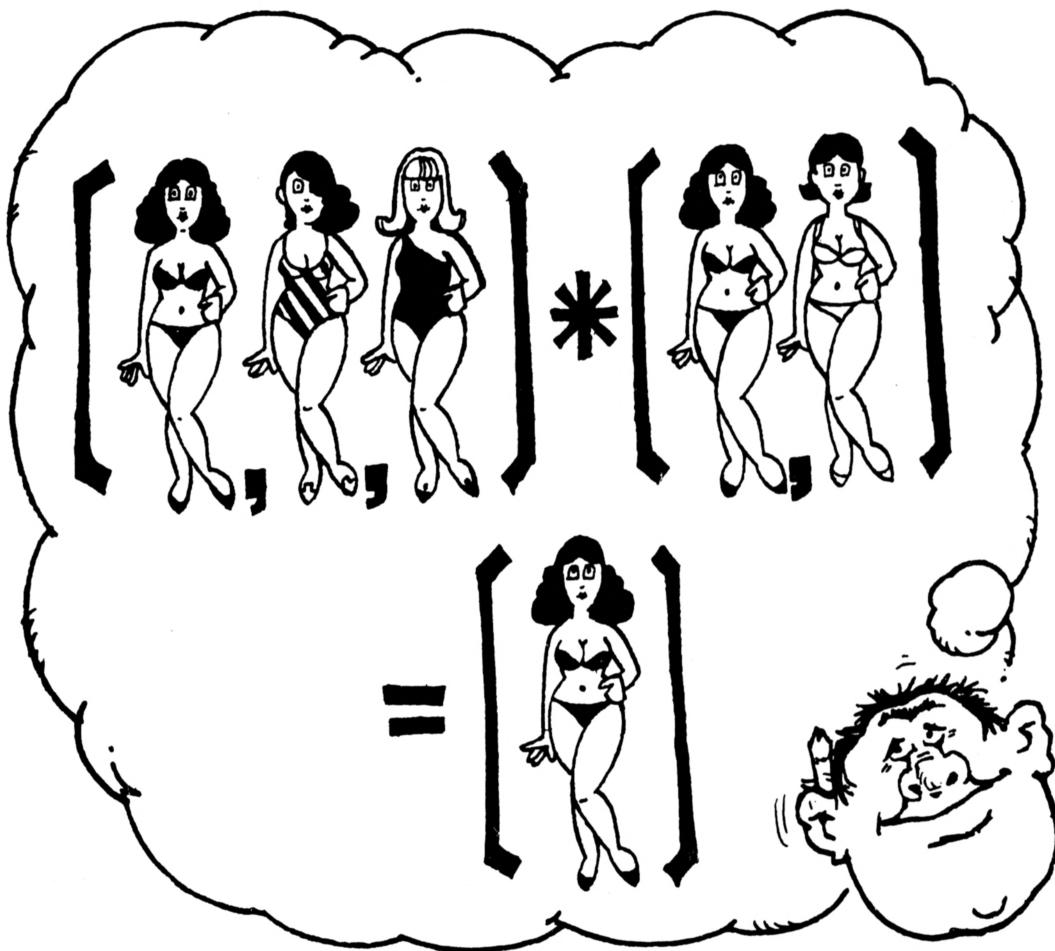
Gráficos

Para mucha gente, usar un ordenador sin capacidad para gráficos es como un árbol sin frutos o una primavera sin flores. El PASCAL no dispone de facilidades para gráficos, ni siquiera de las muy elementales que son necesarias para desplazar un cursor. Como se trata de una omisión tan estrepitosa, son muchos los compiladores que le han agregado sus propias facilidades gráficas no-estándar. En este punto, todo lo que podemos hacer es remitirte a tu manual PASCAL local.

Sumario

Tras la lectura de este capítulo, es probable que te hayas quedado con la impresión de que el sacar o meter algo en tu programa PASCAL es una empresa tan difícil como la de recorrer todo el campo de golf de St. George equipado sólo con un *putter*. Bien, no te dejes abatir demasiado. Usualmente podrás completar finalmente tu tarea, especialmente si tu PASCAL contiene unas ampliaciones razonables, y sobre todo si te construyes tus herramientas propias, que te ayuden para vencer los *bunkers* y fortines del PASCAL.

A pesar de todo, queda la sensación de que tu tarea es mucho más dura de lo necesario.



No piensas más que en los conjuntos.

Reprobación dirigida a un adolescente programador de PASCAL

10 Conjuntos

Tras debatirnos chapoteando en las turbias aguas de la entrada/salida, vamos a volver ahora a una clara y cristalina laguna. En vez de enfrentarnos a un capítulo largo y complicado, empezamos uno breve y (así lo esperamos) sencillo.

Retornamos otra vez al tema de los tipos de datos, y vamos a describir uno nuevo que deberá cambiar tu modo de pensar en relación con algunos de tus programas.

Introducción a los conjuntos

En PASCAL se puede definir un tipo de datos que es un conjunto de objetos de algún otro tipo de datos. A este último se le llama *tipo base* del conjunto. El tipo base de un conjunto puede ser cualquier tipo, excepto el real (de números reales). Usualmente, es un tipo definido por el usuario, un tipo subrango, o *char*.

Aparte del PASCAL, son pocos los lenguajes de programación bien cono-

cidos que soportan conjuntos; e incluso en el PASCAL, algunos compiladores de las primeras hornadas imponían onerosas restricciones sobre el tamaño de los mismos. El resultado de ello es que los conjuntos permanecen ignorados para muchos programadores de PASCAL. Es una lástima, porque los conjuntos pueden ser increíblemente útiles, y contribuir en gran manera a que un programa sea una descripción más próxima y ajustada del problema cuya solución se busca.

Un ejemplo de conjunto

Nuestro primer ejemplo de conjunto tiene como tipo base un tipo definido por el usuario. Este tipo base consiste en una sucesión de nombres de lenguajes de programación. Por tanto, se trata de un conjunto de lenguajes de programación. Se define en esta forma:

```
type
    lenguajes = (Pascal, BASIC, BCPL, SIMULA, ALGOL60, ALGOL68,
                 COBOL);
    lenguajesconocidos = set of lenguajes;
```

Por lo que antecede, puedes ver que para definir un conjunto se escribe **set of** seguido por el tipo base. Una vez hecha esta declaración, puedes usar el tipo de datos en la forma normal y, en particular, puedes declarar variables como de dicho tipo de datos, por ejemplo:

```
var
    lenguajesPrimple: lenguajesconocidos;
    lenguajesMudd: lenguajesconocidos;
    sabelotodo, casosinremedio: lenguajesconocidos;
```

Las variables del tipo de datos *lenguajesconocidos* pueden tomar como valor cualquier subconjunto del conjunto completo. En uno de los extremos, este subconjunto puede ser nulo (el *conjunto vacío*), o, en el otro extremo, puede ser el conjunto de los lenguajes de programación conocidos por una persona determinada.

Cuando se quiera asignar un valor a una variable de *lenguajesconocidos,* se puede usar un *constructor de conjunto.* El constructor de conjunto se escribe haciendo una lista de aquellos elementos que se desee incluir, y encerrando la lista entre corchetes, por ejemplo:

```
lenguajesPrimple: = [Pascal, BCPL];
```

(* *Como los miembros del conjunto se pueden escribir en cualquier orden, lo anterior también se puede escribir así*:
lenguajesPrimple: = [*BCPL*, *Pascal*]; *)
LenguajesMudd: = [*BASIC*];
casosinremedio: = [ ]; (* *un conjunto vacío* *)
sabelotodo: = [*Pascal*, *BASIC*, *BCPL*, *SIMULA*, *ALGOL*60, *ALGOL*68, *COBOL*];

En realidad, existen miles de lenguajes de programación, aunque sólo una docena aproximadamente son los ampliamente conocidos. Si ensancháramos nuestro tipo de datos *lenguajes* hasta que estuviese formado por 100 lenguajes, resultaría muy aburrido escribir el constructor de conjunto que habría de asignarse a *sabelotodo*. Para aliviar esta molestia, el PASCAL permite que los elementos de que se compone un constructor de conjunto puedan especificarse por medio de subrangos del tipo base. Así, podríamos escribir

sabelotodo: = [*Pascal*..*COBOL*];

Como PASCAL es el primer valor del tipo base y COBOL es el último, esto equivale a especificar todos los valores incluidos en el tipo base.

A continuación damos otros dos ejemplos:

sabemás: = [*Pascal*..*BCPL*, *ALGOL*60..*COBOL*]; (* *todos excepto el SIMULA* *)
sabealgunos: = [*Pascal*..*BCPL*, *ALGOL*60, *COBOL*];

Un conjunto se parece algo a una matriz o *array* Booleana, en cuanto que cada posible miembro del conjunto puede estar presente o no; en otras palabras, su presencia puede ser "verdadera" o "falsa". Llevando esta idea un paso más adelante, un conjunto es equivalente a la "cadena de bits" que se encuentra en los lenguajes ensambladores y en algunos lenguajes de alto nivel, es decir, una sucesión de valores 1 (o "verdadero") y 0 (o "falso"). Sin embargo, los conjuntos del PASCAL son mucho mejores que las cadenas de bits, porque facilitan la lectura del programa al desterrar la influenca del Sr. 869704 o, en este caso, la del Sr. 110101000101001000.

Operaciones con conjuntos

Los conjuntos no son ciudadanos de segunda, como algunos tipos de datos del PASCAL. Bien es verdad que no se pueden usar como entrada/salida, pero, al menos, existen algunos operadores que se les pueden aplicar. En especial, se pueden aplicar los operadores "+", "—" y "*". Probablemente, tú

podrás deducir del significado de estos operadores en aritmética lo que los mismos significan cuando se les aplica a conjuntos. Así, el conjunto

$x + y$

es aquél cuyos elementos son elementos de x, o de y, o de ambos. El nombre técnico para esto es la *unión de conjuntos.* Alternativamente, y si persistimos en contemplar los conjuntos como cadenas de bits, un "+" es un "or' lógico (el "*" resultará ser el "and" lógico). En nuestro ejemplo, la unión

lenguajesPrimple + *lenguajesMudd*

es el conjunto

[*Pascal, BCPL, BASIC*]

y la unión de cualquiera de nuestras variables con *sabelotodo* seguirá siendo el conjunto de todos nuestros lenguajes.

El operador menos significa la *diferencia de conjuntos.* Así

$x — y$

es aquel conjunto cuyos elementos pertenecen a x, pero no a y. En nuestro ejemplo,

sabelotodo — *lenguajesPrimple*

es el conjunto

[*BASIC, SIMULA, ALGOL*60, *ALGOL*68, *COBOL*]

Finalmente, el operador "*" significa la *intersección de conjuntos.* Es de tal modo que

$x * y$

es el conjunto cuyos elementos pertenecen a x y a y. En nuestro ejemplo,

lenguajesPrimple * *lenguajesMudd*

es el conjunto vacío. En cambio, la intersección de *sabelotodo* con cualquier conjunto z es idéntica al valor de éste.

Debes ser capaz de pensar ejemplos en los que sean útiles estos operadores aplicados a los conjuntos. En nuestro caso particular, la unión e intersección de *lenguajesconocidos* sería útil para un programa destinado a asignar equipos de programadores a tareas de programación que exigiesen el conocimiento de ciertos lenguajes.

Operaciones de relación con conjuntos

A los conjuntos se les pueden aplicar, además de los "+", "—" y "*", varios operadores de relación. La forma en que éstos funcionan es la siguiente:

- $x = y$ es *verdadera* cuando x e y sean conjuntos idénticos;
- $x <> y$ es *verdadera* cuando x e y no sean conjuntos idénticos;
- $x <= y$ es *verdadera* si todos los elementos de x lo son también de y;
- $x >= y$ es *verdadera* si todos los elementos de y lo son también de x;
- los operadores '>' y '<' no pueden aplicarse a conjuntos.

Dadas estas reglas, la relación

lenguajesPrimple * *lenguajesMudd* = *casosinremedio*

es *verdadera,* porque los dos lados de la ecuación con conjuntos vacíos. Además,

[*BASIC, SIMULA*] = [*SIMULA, BASIC*]

es *verdadera* porque el orden de los elementos es indiferente. Como tercer ejemplo,

sabeloto > =*x*

es *verdadera* para cualquier conjunto de lenguajes x.

Otro ejemplo, cuyo impacto inicial es sorprendente, es que

lenguajesMudd > = *lenguajesPrimple*
y *lenguajesPrimple* > = *lenguajesMudd*

son ambas *falsas.* Así que ten cuidado de no aplicar directamente a las relaciones entre conjuntos tu forma de pensar relativa a las relaciones entre números.

Finalmente, no sólo puedes usar en los conjuntos cuatro de los operadores relacionales existentes, sino que tienes también un nuevo operador: el **in** o "en". Su funcionamiento es tal que

e **in** *x*

es *verdadera* si e es un elemento del conjunto x. El tipo de datos de e tiene que ser igual que el tipo base del conjunto x. En nuestro ejemplo,

BASIC **in** *lenguajesPrimple*

es *falsa,* porque Primple, ciertamente, no admite conocer el BASIC.

Expresiones dentro de constructores de conjuntos

En los ejemplos que hemos dado hasta ahora, los elementos incluidos en los constructores de conjuntos han sido siempre constantes. En realidad, pueden ser cualquier tipo de expresión, como se ve en

```
var
    x: set ot 1..31;
    p,q: integer;
egin
(*  .
    .
    .  *)
    x:= [p,q — 1..q +3, 19, 20];
(*  .
    .
    .  *)
    x:= x + [p..q];
(*  .
    .
    .  *)
```

Si *p* es menor que *q,* entonces [*p..q*] es, desde luego, el conjunto vacío.

Un ejemplo completo con caracteres

Los conjuntos son útiles también para simplificar expresiones Booleanas en las que intervengan muchas operaciones **or.** Vamos a dar un ejemplo de esto en el que se usa un conjunto cuyo tipo base es *char*. Este es, en efecto, uno de los tipos base más populares, y tal vez nuestro ejemplo ponga de manifiesto la razón de ello.

```
program cuentacaracteres(input, output);
(* Cuenta, dentro del texto de entrada:
    1. el número de letras o cifras
    2. el número de caracteres de puntuación ('.' o ',' o ';' o ':') *)

var
    cuentaletrascifras: 0..maxint;
    cuentapuntuación: 0..maxint;
    carácteractual: char;
```

```
begin
    cuentaletrascifras:= 0;
    cuentapuntuación:= 0;
    while not eof(input) do
    begin
        read(carácteractual);
(* En la sentencia que sigue se da por supuesto que el juego de caracte-
    res es tal que todas las letras tengan códigos adyacentes, y lo mismo
    las cifras *)
        if carácteractual in ['a'..'z', 'A'..'Z', '0'..'9'] then
           cuentaletrascifras:= cuentaletrascifras + 1;
        if carácteractual in ['.', ',', ';', ':'] then
           cuentapuntuación:= cuentapuntuación + 1;
    end;
    writlen('El número de letras o cifras es', cuentaletrascifras);
    writeln('El número de caracteres de puntuación es', cuentapuntuación);
end.
```

Este programa no usa variables del tipo de conjunto de datos. Todo lo que usa son constructores de conjuntos y la facilidad **in;** no obstante, tanto una cosa como la otra desempeñan un considerable papel para hacer que el programa sea fácil de escribir y de comprender.

Un segundo ejemplo completo

Un segundo ejemplo completo nos muestra un programa relativo a un tablero de tiro con dardos. Un juego que un extremado aficionado o un borracho consumado pueden completar en un tiempo razonable es el siguiente: hacer blanco en todos los números del 1 al 20, en cualquier orden.

El programa que sigue da entrada a los tanteos o puntuaciones e imprime un mensaje cuando se ha hecho blanco en todos los números.

```
program coberturadardos(input, output);
(* Lee los números constantemente hasta que cada uno de los números
    1 a 20 se haya producido por lo menos una vez *)
var
    númerosacertados: set of 1..20;
    tanteo: 1..50 (* un tablero de dardos contiene también 25 y 50 *)
begin
    númerosacertados:= [ ]; (* inicialmente, el conjunto vacío *)
    repeat
        read(tanteo);
```

```
        if tanteo < = 20 then
            númerosacertados: = númerosacertados + [tanteo];
    until númerosacertados = [1..20];
    writeln('*** ¡MUY BIEN! Ha cubierto todos los números');
end.
```

La línea interesante es la que actualiza a *númerosacertados*. Obsérvese que, cuando el operador "+" se aplica a conjuntos, han de ser conjuntos sus dos operandos. Por consiguiente, el segundo operando ha de ser [*tanteo*], que es un conjunto que contiene un solo elemento, el valor del *tanteo*. La sentencia

númerosacertados: = *númerosacertados* + *tanteo*;

sería un *error,* porque *tanteo* no es un conjunto. No tenemos duda de que tú, como nosotros, te olvidarás de esta regla una y otra vez cuando te pongas a escribir un programa. Sin embargo, tal vez el ejemplo precedente te sirva de recordatorio cuando te preguntes muy intrigado cuál será el error de sintaxis del que se te avisa.

Conjuntos desemejantes

Al igual que, dentro de un solo programa, se puede emplear cualquier número de matrices de distintos tipos y formas, también se puede emplear cualquier número de conjuntos de diferentes tipos base. Así, nuestra variable *lenguajesPrimple* podría aparecer en el mismo programa que *númerosacertados*. Lo que, sin embargo, no se puede hacer es aplicar las operaciones con conjuntos a tipos desemejantes, por ejemplo,

lenguajesPrimple + *númerosacertados*

o

lenguajesPrimple = *númerosacertados*

Tales operaciones carecen de cualquier significado, y Perkins las señalará como errores con toda razón.

Restricciones en relación con los conjuntos

Como ya hemos mencionado antes, algunos de los primeros compiladores PASCAL imponían importantes restricciones sobre el tamaño de los conjun-

tos. Típicamente, un conjunto estaba limitado a un reducido número de elementos, digamos unos 60. Como quiera que casi todos los juegos de caracteres tienen más de 60, esto hacía imposible cualquier **set of** *char*. Frecuentemente, la restricción era aún más abrumadora. No se permitían tipos tales como

x = **set of** 58..62; (* *límite superior* > 60 *)
y = **set of** *'a'..'z'*;

La razón era que los límites de los subrangos han de estar entre 0 y 59, o, en el caso de los caracteres (*char*), los códigos internos han de estar dentro de este margen. La mayoría de los juegos de caracteres no satisface esta restricción impuesta sobre *char*. Si se aplica esta restricción, nuestro ejemplo de verificación de letras y cifras podría ser inutilizable, porque el carácter sometido a verificación y/o los elementos del constructor de conjunto podrían hallarse fuera del margen permitido.

Tenemos la esperanza de haber llegado a tiempos más felices, y de que los viejos compiladores estén siendo sustituidos por otros escritos por gentes que aprecien el valor verdadero que tienen los conjuntos. Sin embargo, es probable que todavía siga habiendo ciertas restricciones sobre el tamaño de los conjuntos, por razones de buena implementación. Es poco probable que se nos vaya a permitir decir

granconjunto = **set of** *integer*;

porque el número de enteros posibles es verdaderamente muy grande. En efecto, es probable que esté prohibido cualquier tipo base que implique enteros negativos o enteros positivos grandes.

Tal vez la divisoria más importante en el tamaño de los conjuntos sea su capacidad para abarcar o no al juego de caracteres, constituido normalmente por 128 ó 256 caracteres. Si los escritores de tus compiladores han hecho el esfuerzo necesario para conseguirlo, no dejes de sacar a sus desvelos todo el fruto posible explotando al máximo los conjuntos y sus posibilidades.

Otras operaciones con conjuntos

"Observo que no hay funciones incorporadas que trabajen sobre conjuntos", dijo Bill. "Por ejemplo, nada que cuente el número de elementos. Por lo que veo, los conjuntos son como la mayor parte de las cosas en el PASCAL: hermosos objetos que son absolutamente ideales, a condición de que no queráis usarlos en programas reales."

Desde luego, no hay funciones incorporadas para los conjuntos, pero en realidad esto es una *buena* característica del PASCAL. Como ya hemos dicho

anteriormente, los lenguajes gigantescos como ballenas no son lo que el usuario necesita.

En las operaciones con conjuntos, el PASCAL es lo bastante breve y elástico como para poder ser utilizado satisfactoriamente. Por ejemplo, el problema de contar los elementos de un conjunto, expuesto por Bill, se puede resolver en la forma siguiente. (Suponemos que el conjunto que se ha de contar es *lenguajesMudd*):

```
count:= 0;
for k:= Pascal to COBOL do
    if k in lenguajesMudd then
        count:= count + 1;
```

Si estamos seguros de que nos esperan penas en el futuro, ¿por qué nos las arreglamos siempre para preparar otras nuevas?

RUDYARD KIPLING

11
Almacenamiento o memoria dinámica

Deficiencias de la memoria estructurada en bloques

En el mundo de los lenguajes estructurados en bloques, del que el PASCAL forma parte, el almacenamiento o memoria empleado para las variables sigue una limpia y ordenada estructura de bloques anidados. Cada vez que se entra en un procedimiento, se reserva espacio de memoria para las variables locales de dicho procedimiento; al producirse el retorno desde ese procedimiento, este espacio de memoria queda liberado. Además de esto, en el programa principal se declara algún almacenamiento, que se halla disponible todo el tiempo; a éste se le denomina el almacenamiento o memoria *global,* porque existe durante todo el tiempo o vida de trabajo del programa. La palabra "global" revela la excluyente concentración del pensamiento de los programadores; el programa en que estamos trabajando en cada momento constituye todo el mundo para nosotros.

Desgraciadamente, los problemas del mundo real no siempre encajan en esta simple disciplina del almacenamiento o la memoria. Consideremos, a título de ejemplo, un programa para el control de los movimientos de aviones en un aeropuerto. Suponemos que el programa se ocupa de tareas tales como:

- control del tráfico aéreo mientras los aviones están en espera para aterrizar;

- asignación de los aviones a las distintas puertas de embarque;
- llevar una lista de los aviones en espera para despegar.

(Tal vez no confíes en tu capacidad y temas que, si fueses tú el responsable de tal tipo de programa, el resultado sería el mayor desastre aéreo de todos los tiempos: cincuenta aviones cayendo a tierra simultáneamente; si te ocurre eso, piensa que el programa es sólo una simulación —tal vez parte de un juego— y no el problema real.)

Cada avión lleva asociadas varias propiedades (por ejemplo, número de vuelo, tamaño del avión, número de pasajeros a bordo y tiempo programado de llegada). Por consiguiente, es natural que cada avión se represente por medio de un registro. Será necesario interconectar los registros de los aviones individuales mediante su inclusión en listas; más adelante estudiaremos los detalles precisos de todo ello. El programa necesitará muchas, como listas de aviones en espera para aterrizar, de aviones en espera para el despegue, de puertas de embarque disponibles, de vuelos programados que tienen retraso, de vuelos *charter* especiales, y tal vez incluso listas de pasajeros. Todas estas listas tienen dos rasgos comunes;

- son *dinámicas* —crecen y se contraen;
- son globales —existen durante todo el funcionamiento del programa.

Considera la forma en que podrías representar estas listas usando las facilidades del PASCAL que ya te hemos descrito. Está claro que cada lista ha de representarse como un *array* o matriz de registros. El problema surge cuando se trata de determinar el tamaño de cada *array*. Este tamaño ha de ser una constante fijada de antemano; aun cuando tu PASCAL te permita declarar una matriz, con una variable como límite superior, por ejemplo,

array[1..*n*] **of** *aviones*;

ello no servirá de mucha ayuda, ya que sigue siendo necesario expresar el tamaño de la matriz (el valor de *n* en nuestro ejemplo) antes de asignarlo.

Si tienes que fijar el tamaño de tus *arrays* o matrices por adelantado, has de hacerlos tan grandes como sea posible. Sería bastante desafortunado que tuvieras que decir a un avión que llegara al aeropuerto: "Lo siento: no cabe usted en mi sistema de control de tráfico aéreo porque mi matriz está llena." Sin embargo, si haces grandes todas tus matrices, se te acabará la capacidad de memoria; en la práctica, los tamaños de tus matrices habrán de ser limitados, y lo de que una matriz se llene es una posibilidad real.

La gran desventaja de un esquema como el precedente, que utiliza matrices o *arrays* separados, es que si uno de ellos llega a llenarse, es muy probable que haya algunos de los otros que dispongan de abundante capacidad no utilizada; por ejemplo, si hay una saturación de aviones, es probable que exista una falta de pasajeros en espera y de puertas de embarque libres. Desgraciadamente, no hay forma de aprovechar esta memoria no utilizada.

Un problema análogo

Para ver en qué forma se puede resolver este problema, vamos a considerar una analogía. Tenemos un grupo de personas que utilizan los servicios de una biblioteca de préstamo de libros. Por cada libro que te lleves, necesitas un vale o *ticket*. A tu familia se le han adjudicado veinte. Preguntas a cada miembro de tu familia cuál es el número máximo de libros de la biblioteca que necesitará tener en préstamo en cualquier momento. El total asciende a treinta y cinco. Por tanto, si en el seno de tu familia tratas de asignar *tickets* a cada persona de ella con arreglo a sus necesidades máximas, se te terminarán los vales de la biblioteca.

Resuelve el problema observando que es muy poco probable que todos los individuos de la familia utilicen su capacidad máxima de préstamo de libros al mismo tiempo. Por consiguiente, con los veinte vales o *tickets* de préstamo constituyes un fondo común que compartirá toda la familia; los miembros de ella tomarán un *ticket* de dicho fondo cuando lo necesiten, y lo devolverán a él cuando hayan terminado. Lo más probable es que los veinte *tickets* resulten suficientes para cubrir la necesidad aparente de treinta y cinco.

La pila

Por tanto, una solución para el problema del almacenamiento dinámico que necesitamos para nuestras listas consiste en poner en un solo fondo toda la capacidad de almacenamiento o memoria. Inicialmente, este fondo de memoria está *libre*, es decir, no asignado. Cuando es necesario añadir un nuevo elemento a una de las listas, se toma ''en préstamo'' una parte de esta memoria libre, y se la usa para contener al elemento de la lista. Si, en una etapa posterior, se deja de necesitar este elemento de lista (por ejemplo, cuando un avión sale del sistema), el espacio de memoria correspondiente a él se puede ''dejar libre'' y devolverlo al fondo de memoria libre. Las ventajas de este esquema son las siguientes:

- sólo se usa memoria cuando es realmente necesario. Por tanto, sólo se agotará la memoria del sistema si toda ella se ha utilizado verdaderamente;

- la memoria que quede libre en una lista podrá ser utilizada de nuevo más tarde, por la misma lista o por otra; incluso, acaso, por una lista con un tipo de datos diferente.

El PASCAL proporciona precisamente un mecanismo de esta clase. Permite mantener estructuras de datos separadas, normalmente de tipos de datos

independientes, todas las cuales comparten el mismo fragmento o espacio de memoria. La memoria no viene y se va cuando entramos y salimos en los procedimientos, sino que está controlada por instrucciones específicas para tomar espacio prestado y para dejarlo libre. Por consiguiente, puedes utilizarla para objetos que sean globales de tu programa. Al espacio de almacenamiento se le llama el *montón* o pila (en inglés, *the heap*) porque es un objeto amorfo, que no se usa en un forma estructurada en bloques. Cuando empezamos la ejecución de un programa PASCAL, toda la memoria principal de nuestra máquina que está sin utilizar se reúne en la pila. (Normalmente, la pila no está relacionada con la memoria de reserva, o *backup*). Una variable a la que se asigne espacio de almacenamiento desde la pila, recibe el nombre de *variable dinámica*. (Emplearemos el término "variable ordinaria" para describir la clase de variables de las que nos hemos ocupado en forma exclusiva antes de este capítulo, es decir, las variables declaradas bajo el encabezamiento **var**).

La utilización de la memoria dinámica

Cuesta un esfuerzo bastante considerable aprender y dominar las facilidades del PASCAL para el almacenamiento dinámico. Ello no es porque estas facilidades sean confusas o indebidamente complicadas —más bien al contrario—, sino porque implican un número de conceptos nuevos.

El más importante de ellos es que las variables dinámicas no se declaran en la sección **var** del programa. De hecho, *las variables dinámicas carecen completamente de nombres*. En su lugar, se accede a ellas por medio de *punteros*.

Punteros

Llegamos ahora al último de los tipos de datos del PASCAL. Su nombre es el *puntero*. Una variable del tipo puntero se llama simplemente un puntero del mismo modo que una variable del tipo *array* se llama un *array* o matriz; puede que esto te suene un poco confuso, pero no lo es en la práctica. Un puntero apunta a una variable dinámica. Los dos son completamente interdependientes en cuanto que:

1) un puntero sólo puede apuntar a una variable dinámica. No puede, por ejemplo, apuntar a una variable ordinaria;

2) la *única* forma en que se puede hacer referencia a una variable dinámica es por medio de un puntero.

En nuestro ejemplo, las variables dinámicas son los registros asociados con aviones, puertas, etc., que constituyen los elementos de nuestras listas. Cada variable dinámica necesitará un puntero asociado con ella.

En términos de implementación, un puntero es simplemente la dirección del byte (o de la palabra) en que empieza un fragmento de almacenamiento dinámico. Es posible que ya estés familiarizado con un concepto similar en programación en lenguaje ensamblador. En ese caso, puedes pensar en el puntero como en algo que puede apuntar a *cualquier* byte de la memoria o almacenamiento principal de tu ordenador. Entonces, la restricción expresada en 1) más arriba puede resultarte sorprendente. La finalidad de la misma, como veremos, es mantener a los programas dentro de una segura disciplina, y vedar los picarescos trucos que son posibles en lenguaje ensamblador.

Los tipos de datos del almacenamiento dinámico

En PASCAL, todas las variables, sean dinámicas o no, tienen un tipo de datos asociado. A las variables ordinarias se les da un tipo de datos cuando se declaran en la sección **var**. Esto es algo que no se puede hacer con las variables dinámicas, puesto que carecen de nombre y no se las declara. En lugar de ello, se adosa el tipo de datos al puntero que apunta a la variable dinámica en cuestión. Las variables dinámicas son normalmente registros, por razones que ya veremos, de modo que este tipo de datos suele describir una estructura de registro. Un puntero se declara poniendo el símbolo "↑" como prefijo del tipo de datos al que ha de apuntar. (El tipo de datos ha de estar representado por un identificador; es la misma regla que se aplica al tipo de datos de un parámetro.) A continuación damos unos ejemplos de declaraciones, en las que suponemos que *avión* y *puerta* se declaran como registros bajo el encabezamiento **type**:

```
type
    punteroavión = ↑ avión;
    punteropuerta = ↑ puerta;
var
    aviónqueaterriza: punteroavión;
    aviónquedespega: punteroavión;
    próximapuertalibre: punteropuerta;
```

Aquí, *aviónqueaterriza*, *aviónquedespega* y *próximapuertalibre* son los tres punteros. El tipo asociado con un puntero es una característica vital del PASCAL. Ello significa que Perkins puede verificar el tipo de las variables dinámicas en la misma forma que en las ordinarias. Así, si cometes en tu programa alguna tontería, como hacer referencia a un avión cuando quieres decir una puerta, el compilador dará un aviso de error. Esto contrasta con lo que sucede

en algunos lenguajes, en los que tal tipo de error puede causar una corrupción de los datos que puede ser catastrófica, y que no se le notifica al usuario.

Asociada con el tipo de datos del puntero hay una constante llamada **nil**. Un puntero puede apuntar a una variable dinámica, o tener el valor **nil**. Si en este momento no hay ningún avión aterrizando, se puede hacer la siguiente asignación:

aviónqueaterriza: = **nil**;

Los punteros son uno de los tipos de datos del PASCAL que no tienen muchos operadores asociados. Todo lo que podemos hacer es asignar un puntero a otro (incluyendo el caso de pase de parámetros), por ejemplo:

aviónquedespega: = *aviónqueaterriza*; (* *aterrizaje abortado* *)

o comparar los valores de dos punteros empleando los operadores relacionales "=" o "< >", por ejemplo:

if *aviónqueaterriza* = *aviónquedespega* **then** (* ... *)

En ambos casos, asignación y comparación, los dos punteros que intervienen han de tener el mismo tipo de datos asociado. Si se pudiera asignar a un puntero un valor de cualquier tipo de datos asociado, la tarea de Perkins sería imposible, porque no sabría a qué estabas apuntando. (En la jerga de los ordenadores, se dice que el PASCAL es *strongly typed,* o con fuerte influencia de los tipos; los tipos de datos de las variables, incluso los de aquellas que se direccionan indirectamente, se conocen de antemano.)

Toma en préstamo y devolución o liberación

Supongamos que deseas tomar en préstamo algún espacio de almacenamiento. Un avión que llega acaba de entrar en el área y deseas crear un registro para él y añadirle a la lista de aviones en espera para el aterrizaje. El espacio de almacenamiento o memoria se toma prestado por medio del procedimiento incorporado *new,* o "nuevo". Este procedimiento toma como argumento suyo un puntero. Su acción es tomar un fragmento de memoria, y hacer que el puntero apunte a él. El tamaño del fragmento de la memoria es el suficiente para contener una variable dinámica del tipo de datos asociado con el puntero.

A continuación se da un ejemplo de llamada de *new:*

var
llegada: apuntdeaparato;

```
begin
(*  .
    .
    .  *)
  new(llegada);
(*  .
    .
    .  *)
```

Hay una forma de *new* o "nuevo" más elaborada, que se puede emplear para economizar un poco de espacio cuando un registro tiene un campo de variantes. No nos preocuparemos de esto. Una llamada de *new* no asigna ningún valor al almacenamiento o memoria que adjudica, y así, lo primero que ha de hacer el programa con el nuevo almacenamiento es poner en él algunos valores. Pronto veremos la forma en que se hace.

El procedimiento complementario del *new* es el *dispose,* que libera un fragmento de memoria. Este también toma como argumento a un puntero, por ejemplo:

dispose(aviónquedespega);

Vale la pena hacer unas cuantas advertencias acerca de *new* y *dispose.* En primer lugar, no existe (normalmente) ninguna comprobación de seguridad contra el problema del puntero colgante (*dangling pointer*), en el cual se libera algún fragmento o espacio de almacenamiento y luego se sigue haciendo referencia a él; ni hay tampoco comprobación alguna contra el error de liberar dos veces inadvertidamente el mismo espacio de memoria. Estos dos errores pueden producir un comportamiento imprevisible del programa. Es uno de los pocos ejemplos del PASCAL en que es posible hacer algo en contra de las reglas sin que le cojan a uno.

En segundo lugar, existe cierto número de compiladores PASCAL que no se comportan en la forma normal en lo que concierne al *dispose.* Algunos no hacen caso alguno de ese procedimiento, mientras que otros lo sustituyen por un mecanismo que hace que el montón se comporte como una pila auténtica o *stack.* Así pues, los programas que usan el *dispose* no son tan portátiles como debieran.

Referencias a las variables dinámicas

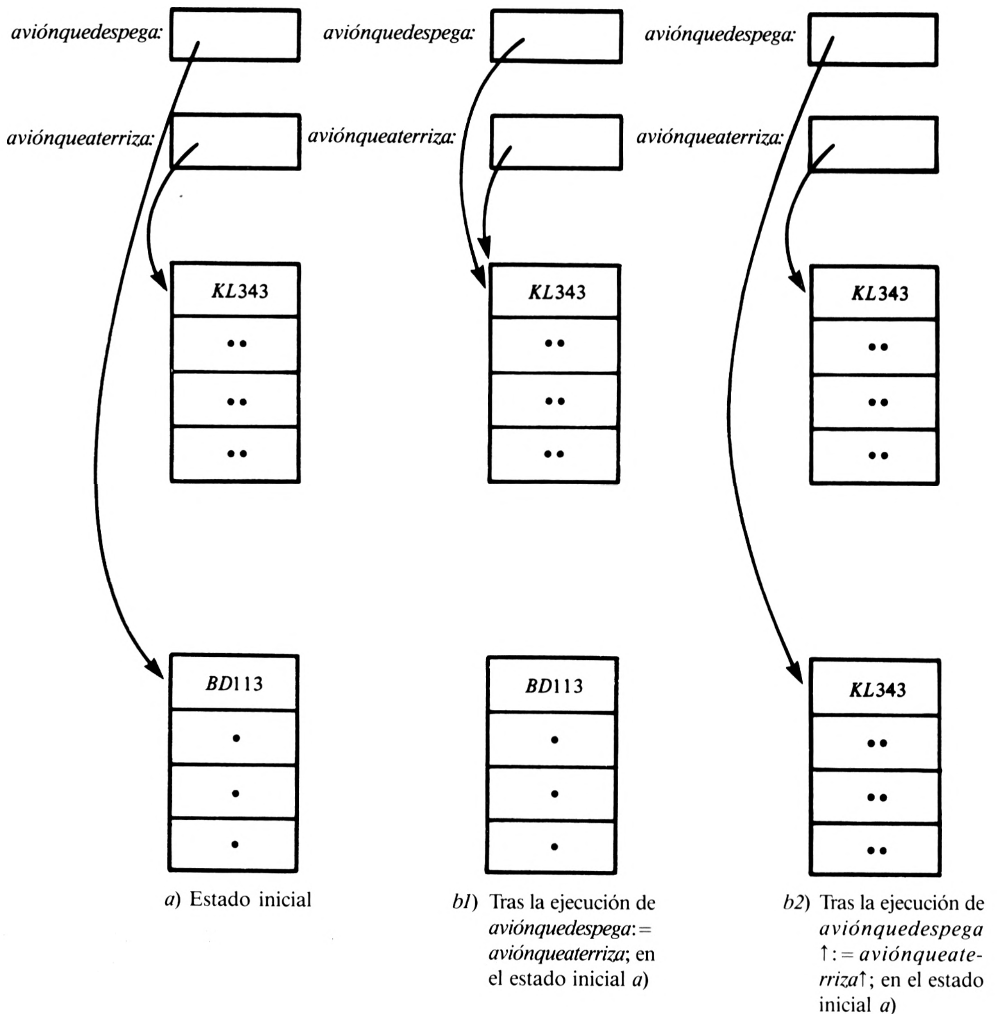
Como ya hemos dicho, una variable dinámica carece de nombre. Para hacer referencia a ella, se escribe el nombre del puntero que apunta a la variable y se le pone *después* el signo "↑". Así, cuando se *declara* un puntero, se pone la flecha "↑" delante del tipo de datos, y cuando se le *usa,* se pone la flecha "↑" detrás de él; indudablemente, esto ha de estar basado en una profunda

lógica, pero, sin embargo, esta regla es una verdadera fuente de triviales errores de sintaxis.

Las variables dinámicas se pueden usar en cualquier parte en que sea posible utilizar una variable ordinaria del mismo tipo de datos, por ejemplo:

aviónquedespega↑ = *aviónqueaterriza*↑;

o, si *aviónaparcado* es una variable ordinaria del tipo *avión,* se puede decir



a) Estado inicial

b1) Tras la ejecución de *aviónquedespega*: = *aviónqueaterriza*; en el estado inicial *a*)

b2) Tras la ejecución de *aviónquedespega* ↑: = *aviónqueaterriza*↑; en el estado inicial *a*)

Figura 11.1. Efectos de las asignaciones utilizando punteros.

aviónaparcado: = *aviónqueaterriza*↑;
(* .
.
. *)
aviónquedespega↑: = *aviónaparcado*;

Para familiarizarnos mejor con los punteros, será valioso considerar la diferencia entre las dos sentencias alternativas

(* 1 *) *aviónquedespega*: = *aviónqueaterriza*;
(* 2 *) *aviónquedespega*↑: = *aviónqueaterriza*↑;

El primer caso es una asignación de un puntero a otro; se hace que el puntero *aviónquedespega* apunte a la misma variable dinámica que *aviónqueaterriza*. Así, cualquier referencia subsiguiente a *aviónquedespega* producirá la misma variable dinámica que una referencia a *aviónqueaterriza*. El segundo caso es una asignación de un registro a otro; el registro almacenado en la variable dinámica *aviónqueaterriza* se copia en la variable dinámica *aviónquedespega*. El resultado es dos copias separadas de la misma cosa. Desde luego, cualquier cambio subsiguiente en una de las copias no afecta a la otra. La figura 11.1 muestra en forma gráfica la diferencia entre los dos casos. Inicialmente, *aviónqueaterriza* apunta al registro correspondiente al avión *KL*343, y *aviónquedespega* apunta al registro correspondiente al avión *BD*113.

Es posible que hayas notado la similitud entre esta notación para los apuntadores y la correspondiente al uso de ventanas en archivos. Tal similitud no es accidental; puedes considerar una ventana como un puntero que señala a un archivo.

Cuando se trata de un registro, es más frecuente la referencia a campos individuales que al registro completo. Si el registro es una variable dinámica, se sigue haciendo referencia a los campos en la forma normal, es decir, añadiendo los nombres de los campos. Así, si *númerodevuelo* es un campo de *avión,* haremos referencia al *númerodevuelo* de la variable dinámica a la que apunta *aviónqueaterriza* en la forma siguiente:

aviónqueaterriza↑.*númerodevuelo*

Sumario intermedio

Ahora ya hemos tratado todos los conceptos nuevos que necesitas conocer. Esencialmente, todo lo que necesitas comprender son los punteros, y los procedimientos *new* y *dispose.*

Hace falta practicar bastante para llegar a ser realmente diestro en el alma-

cenamiento dinámico. La causa es, sin duda, que el nivel adicional de tortuosidad que suponen los apuntadores es una pequeña carga más para la mente. Más adelante, en este mismo capítulo, mostraremos algunos ejemplos más largos, lo cual podrá contribuir a que los conceptos queden más claros.

Las estructuras de datos figuran entre los objetos más importantes del trabajo con ordenadores. Esta expresión se emplea para abarcar a cualquier colección de elementos de datos relacionados entre sí. Así, los *arrays* o matrices y los registros son estructuras de datos. Sin embargo, cuando las estructuras de datos llegan a ser verdaderamente interesantes es cuando las variables dinámicas se enlazan o articulan con los punteros.

Todo este conjunto constituye un área fascinante para el estudio y la experimentación. Hay multitud de buenos libros que leer; uno de ellos es *Data structure techniques* (*Técnicas de estructura de datos*), por T.A. Standish (1980). Si realmente aspiras a la profesionalidad en materia de ordenadores, has de leer la más grande obra de esta ciencia: *The art of computer programming* (*El arte de la programación de ordenadores*), por Donald Knuth (1973). No te dejes asustar por las matemáticas que hay al principio; más adelante viene un abundante material que es eminentemente legible, aun cuando no domines las matemáticas. En particular, el volumen 1 contiene mucha y buena información acerca de las estructuras de datos.

Apenas es posible leer una descripción de un trabajo avanzado de *software* sin encontrarse con un esquema de una elaborada estructura de datos. En la figura 11.2 tenemos un ejemplo de este tipo.

La moraleja, pues, es perfectamente clara: a medida que vas pasando a tareas de programación más avanzadas, se va haciendo cada vez más importante el tener un buen dominio de las estructuras de datos. En realidad, tu mismo éxito puede depender de que elijas la estructura adecuada para la tarea de que se trate.

Una lista como, por ejemplo, una de nuestras listas de aviones, es en realidad un ejemplo bastante sencillo de una estructura de datos. Sin embargo, como éste no es un libro sobre estructuras de datos, nos limitaremos en él a este caso sencillo, esperando que lo domines y que después te sientas preparado para empresas más serias.

Variables dinámicas enlazadas (*linked*)

La misma finalidad del almacenamiento dinámico significa que el número de variables dinámicas en existencia en cualquier momento es algo que no se puede fijar de antemano. Por tanto, no se puede pensar siquiera en declarar, dentro de la sección **var**, un puntero que corresponda a cada variable dinámica. La única forma posible de hacerlo sería tener una matriz o *array* de punteros, pero tal matriz habría de tener un tamaño máximo fijo, lo cual destruiría en gran medida el objetivo perseguido.

La forma en que se resuelve este problema consiste en organizar a las variables dinámicas en una estructura de datos, cada uno de cuyos elementos contenga uno o más punteros a las otras variables dinámicas relacionadas con él. Necesitamos un puntero para acceder a la "primera" variable dinámica de la estructura. Todos los restantes elementos se pueden encontrar siguiendo una cadena de punteros que empieza con este primero.

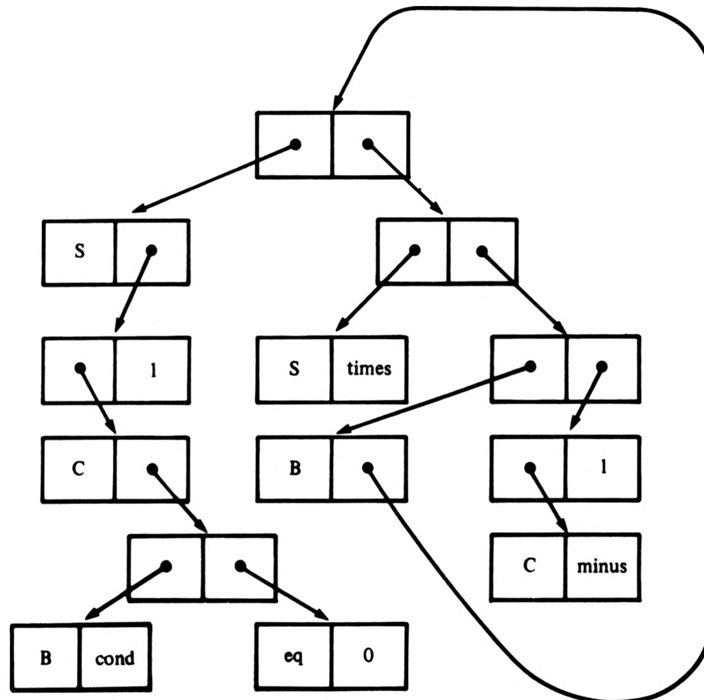


Figura 11.2. Ejemplo de una estructura de datos. (Reproducido con autorización de John Wiley and Sons de un artículo técnico por Turner, 1979.)

Con nuestra sencilla estructura de datos —la lista— es fácil conseguirlo. Cada elemento de la lista es una variable dinámica que representa un registro, y dentro de éste reservamos un puntero que se usa para apuntar al elemento siguiente de la lista. Si el elemento actual de la lista es el último de ella, el puntero tendrá el valor **nil.** Con cada lista está asociada una variable ordinaria, que se llama la *cabeza* de la lista y apunta al primer elemento. Si una lista está vacía o nula, la cabeza tendrá el valor **nil.** Usualmente, la cabeza de la lista se declarará en forma global, ya que no ha de desaparecer mientras la lista esté en existencia.

En la figura 11.3 vemos una lista de tres aviones, *BA*136, *KL*026 y *VA*113. El puntero *cabezadelista* constituye la cabeza de la lista. (Las posiciones en que hemos dibujado los bloques de memoria no importan en absoluto; cualquier disposición de estructura equivalente tendría el mismo significado.)

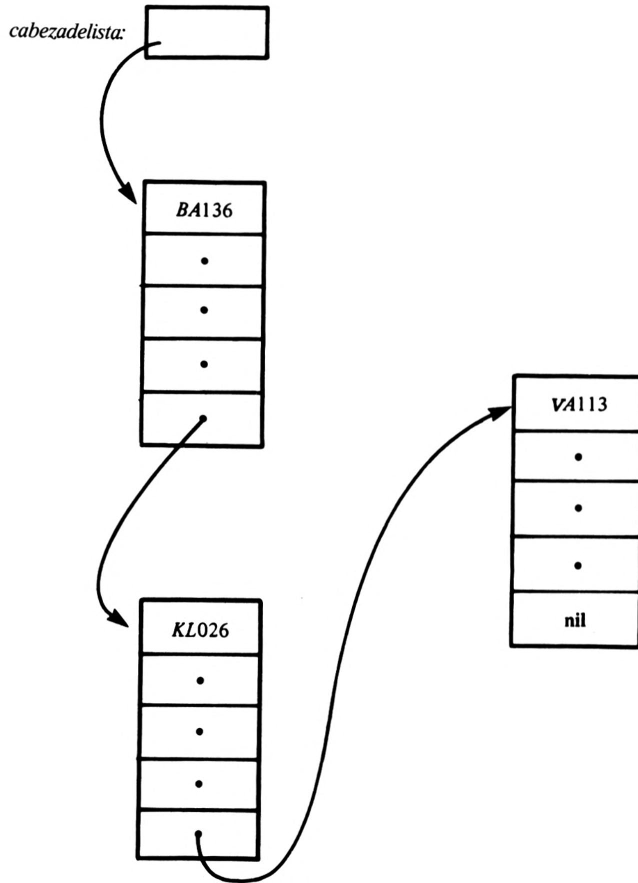


Figura 11.3. Una lista de tres aviones.

Hemos dicho antes que normalmente las variables dinámicas suelen ser registros. Esperamos que ahora veas por qué: cada variable dinámica es usualmente una combinación de valores, algunos de ellos conteniendo información acerca del objeto que la variable dinámica representa, y otros (en nuestro caso) actuando como punteros a otras variables dinámicas subsiguientes.

Estilo de programación

Hemos insistido en que con cada variable dinámica debe haber asociado por lo menos un puntero. Si no hubiera ninguno, no podríamos hacer referencia a la variable, ni siquiera *deshacernos* (*dispose*) de ella. (Algunos lenguajes disponen de ''recogida de basuras'', mediante la cual se deshacen automática-

mente de las variables no referenciadas; el PASCAL no lo hace, si exceptuamos que toda la memoria se libera al final mismo de cada ejecución, de modo que la próxima se inicie con la memoria otra vez virgen.)

El estilo normal de programación es tener un puntero asociado con cada variable dinámica para hacer referencia a ella y para enlazarla a una estructura de datos, más un número de ''punteros errantes'' (*roving pointers*) utilizados en el interior del programa para hacer referencia a las variables dinámicas que son de interés actual; por ejemplo, el avión que esté tomando tierra en este momento.

Ejemplo de procedimientos de tratamiento de listas

Dedicaremos el resto de este capítulo a algunos ejemplos concretos de fragmentos de programa para manipular listas de aviones. Esperamos que por este medio te familiarices más con el almacenamiento dinámico.

En primer lugar, declaramos los tipos de datos necesarios, cosa que se hace en la forma siguiente:

```
type
    complemento = 0..1000; (* número de pasajeros en un avión *)
    vuelo = packed array [1..5] of char; (* una cadena de 5 caracteres *)
    apuntdeaparato = ↑avión;
    avión =
        record
            númerodevuelo: vuelo;
            pasajeros: complemento;
            (*  otros campos
                .
                .
                .  *)
            siguiente: apuntdeaparato;
        end;
```

El apuntador al siguiente avión no tiene que ser forzosamente el último campo del registro *avión,* pero, por convención, las listas se definen normalmente en esta forma.

Obsérvese que hemos dado un nombre, *apuntdeaparato,* al tipo de datos constituido por un indicador a un avión. Este nombre hará falta cuando escribamos procedimientos que tomen como parámetros a tales punteros; recuérdese que el tipo de un parámetro ha de ser un tipo de datos provisto de nombre.

Puesto que tenemos un nombre, será bueno usarlo en todo lugar en que sea posible. Si tu compilador insiste en la coincidencia de nombres, un parámetro *apuntdeaparato* concuerda solamente con un argumento *apuntador-*

aparato y no con un argumento, aparentemente idéntico, ↑*avión*. Así pues, hemos usado *apuntdeaparato* en el campo *siguiente* de *avión*.

El ejemplo pone de manifiesto un punto interesante. La referencia ↑*avión* viene antes que la definición de *avión*. (Si hubiéramos puesto la definición de *apuntdeaparato* después de *avión*, ello habría dado lugar a un error cuando se produjera la existencia de *indicadoraparato* dentro de *avión*.) Esta situación, en la que el nombre de un tipo de datos sigue al símbolo "↑", es el único caso en PASCAL en que se puede usar un identificador antes de haberlo declarado. Sin embargo, el identificador ha de ser declarado dentro de la misma sección **type**. El objeto de esta suavización de las reglas normales es para que se puedan definir estructuras de listas, tales como nuestro *avión*.

La impresión de una lista

Nuestro primer ejemplo expone un procedimiento que avanza secuencialmente a lo largo de una lista. Esos procedimientos son muy corrientes. La tarea específica del procedimiento de nuestro ejemplo es imprimir el *númerodevuelo* de cada avión incluido en la lista. Es como sigue:

```
procedure escribevuelos(apuntdelista:apuntdeaparato);
(* Escribe los números de todos los vuelos incluidos en la lista de aviones
   dada *)
var
   apuntactual: apuntdeaparato; (* empleado para apuntar sucesiva-
                                    mente a cada avión *)
begin
   apuntactual:= apuntdelista;
   while apuntactual< >nil do
   begin
      writeln(apuntactual↑.númerodevuelo);
      apuntactual:= apuntactual ↑.siguiente; (* pasa al avión siguiente *)
   end;
end; (* de escribevuelos *)
```

El procedimiento precedente nos muestra un mecanismo típico para el tratamiento de listas. El bucle **while** recorre una lista en forma sucesiva hasta que se encuentra un puntero **nil**. Cada vez que se recorre el bucle, se actualiza un puntero (*apuntactual*) para pasar al elemento siguiente de la lista. Obsérvese que este procedimiento funciona correctamente en una lista nula, así como en una lista parcial, es decir, el caso en que el argumento apunta a un lugar intermedio de una lista en lugar de hacerlo a la cabeza de la misma.

El profesor Primple prefiere escribir el procedimiento en una forma repetitiva o recursiva, a saber:

```
procedure profescribe(apuntdelista: apuntdeaparato);
(* Especificación igual a la de escribevuelos, pero éste funciona en
   forma repetitiva *)
begin
   if apuntdelista < > nil then
   begin
      writeln(apuntdelista ↑.númerodevuelo);
      profescribe(apuntdelista ↑.siguiente);
   end;
end; (* profescribe *)
```

Sin embargo, si el profesor llegara alguna vez a ejecutar realmente sus programas, hallaría que este elegante procedimiento repetitivo no es muy práctico. Si la lista contiene 1.000 elementos, *profescribe* se llama a sí mismo repetitivamente hasta una profundidad de 1.000 llamadas. Cada llamada ocupa espacio dentro del ordenador, con lo que en un ordenador pequeño el resultado probable es una abrupta interrupción de la impresión hacia la mitad de la lista.

A pesar de ello, vale la pena mostrar la creación del profesor. Demuestra la forma en que la repetición puede simplificar la estructura lógica de un procedimiento mediante la eliminación de algunos de los cansados detalles de *housekeeping* (preparación o acondicionamiento). En nuestro caso, se elimina el puntero *apuntactual* que se desplaza a lo largo de la lista. Para estructuras de datos más complicadas que las listas, y en casos en los que no surja una repetición muy profunda, los procedimientos repetitivos son sencillos y eficaces.

Adición de elementos a una lista

El procedimiento que sigue añade un nuevo avión a una lista. El espacio de almacenamiento o memoria para el nuevo avión se toma en préstamo en forma dinámica. El campo *siguiente* del avión se pone en el punto al que apunta el parámetro *apuntinserción*, y el mismo *apuntinserción* se actualiza para que apunte al nuevo avión:

```
procedure agregaalista(var apuntinserción: apuntdeaparato; f: vuelo; c:
                       complemento);
(* Crea el registro para un nuevo avión, conteniendo el vuelo y el com-
   plemento correspondientes, y lo agrega a la lista en un punto anterior
   al avión que hay en apuntinserción *)
var
   aptr: apuntdeaparato; (* un apuntador temporal *)
begin
   new(aptr);
```

```
(* Ahora pone los valores en el nuevo registro *)
    aptr ↑.númerodevuelo: = f;
    aptr ↑.pasajeros: = c;
    aptr ↑.siguiente: = apuntinserción; (* añade el resto de la lista original después del nuevo avión *)
    apuntinserción: = aptr (* enlaza el nuevo avión a la lista *)
end; (* agregaalista *)
```

Obsérvese que es necesario escribir **var** antes del parámetro *apuntinserción* porque el procedimiento cambia el valor de este puntero, para hacer que apunte al elemento recién insertado. Este procedimiento se puede utilizar para insertar un elemento en cualquier lugar de una lista, sea en la cabeza, al final, o en un punto intermedio. La siguiente secuencia de llamadas muestra la construcción de una lista:

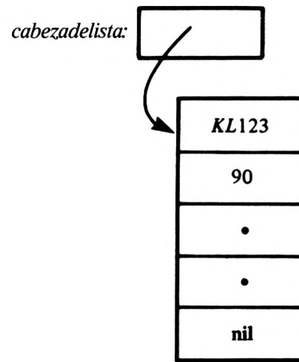
```
    cabezadelista: = nil; (* la lista es nula inicialmente *)
    agregaalista(cabezadelista, 'KL123', 90); (* pone en la lista al primer avión *)
    agregaalista(cabezadelista,'BR456', 110); (* lo añade en la cabeza de la lista, delante de KL123 *)
(* En este punto, la lista consta de BR456 seguido de KL123 *)

    agregaalista(cabezadelista ↑.siguiente,'IC001', 0);
(* El IC001 se agrega después del BR456 y antes del KL123 *)
```

Una vez ejecutadas las tres llamadas de *agregaalista,* la lista está formada por *BR*456, seguido de *IC*001, al que a su vez sigue *KL*123. La forma en que se construye se muestra en la figura 11.4.

La siguiente secuencia de código es, de alguna manera, más general. Agrega un nuevo elemento (*LA*789) al final de la lista a la que apunta *cabezadelista,* independientemente del número de aviones que existan ya en ella.

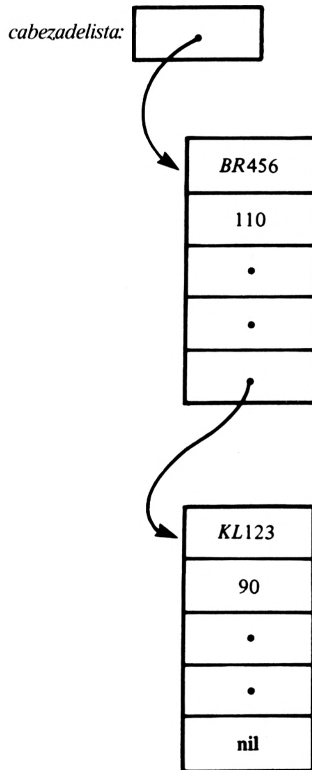
```
if cabezadelista = nil then
   agregaalista(cabezadelista, 'LA789', 99)
else
begin
   apuntactual: = cabezadelista;
   (* Halla el final de la lista *)
   while apuntactual ↑.siguiente < > nil do
      apuntactual: = apuntactual ↑.siguiente;
   agregaalista(apuntactual ↑.siguiente, 'LA789', 99);
end;
```

Superficialmente, el código precedente es excesivamente complicado. Mucho más sencillo es

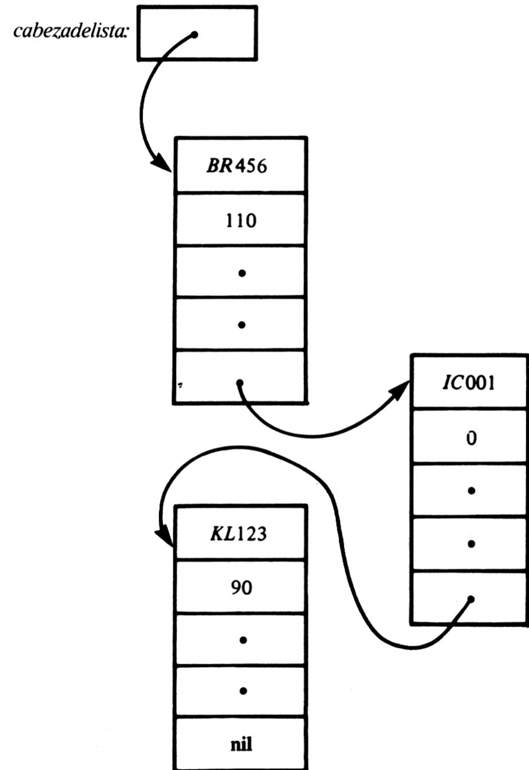


a) inicialmente

b) después de la adición de *KL*123



c) después de la adición de *BR*456



d) después de la adición de *IC*001

Figura 11.4. Creación de una lista.

```
apuntactual:= cabezadelista;
while apuntactual < > nil do
    apuntactual:= apuntactual ↑.siguiente;
agregaalista(apuntactual, 'LA789', 99);
```

El único problema que tiene este sencillo código es que no funciona. La razón es que *agregaalista* cambia el valor de su parámetro **var**. La intención de este cambio es alterar un puntero dentro de la lista donde ha de hacerse la inserción. Nuestro sencillo código no cambia la lista en ninguna forma; todo lo que hace es actualizar el valor del puntero temporal *apuntactual* de modo que apunte al nuevo elemento. Así que necesitamos el código más complicado, que pasa como argumento o la cabeza de la lista o bien un puntero en el interior de la lista que queremos cambiar. Si hallas que esta expresión es complicada, trata de resolver el problema tú mismo.

''Sólo es complicado porque no aprovecháis las ventajas de la repetición'', dijo el profesor. ''Todo lo que necesitáis es un procedimiento así

```
procedure agregaalfinal(var aparatos: apuntdeaparato; f: vuelo; c: complemento);
(* crea un nuevo avión y lo agrega al final de la lista *)
begin
    if aparatos = nil then
        agregaalista(aparatos, f, c)
    else
        agregaalfinal(aparatos ↑.siguiente, f, c);
end; (* de agregaalfinal *)
```

y a continuación escribís

```
    agregaalfinal(cabezadelista, 'LA789', 99);
```

Así de sencillo.'' Un punto para Primple.

Para borrar o suprimir (*delete*) de una lista

En nuestro ejemplo final se borra o suprime un avión de una lista, ilustrando así el uso del procedimiento *dispose*. El procedimiento correspondiente es el que sigue:

```
procedure suprimedelista(var apuntdelista: apuntdeaparato);
(* Suprime de una lista el avión al que apunte apuntdelista; apuntde-
   lista es el campo 'siguiente' del avión al que se ha de borrar. Cuando
   se haya de borrar el primer avión, apuntdelista es la cabeza de la lista *)
```

```
var
    enlace: apuntdeaparato; (* campo 'siguiente' del avión borrado *)
begin
    if apuntdelista = nil then
    begin
        writeln('*** Error: no se puede borrar o suprimir nada de una lista
                nula');
        (* ... Facilite información adicional... *)
        halt; (* ..o, mejor, llame a alguna rutina de recuperación.. *)
    end;
    enlace: = apuntdelista↑.siguiente; (* apunta al avión siguiente al que
                                           ha de suprimirse *)
    dispose(apuntdelista);
    apuntdelista: = enlace (*  actualiza para apuntar al avión siguiente  *)
end; (* suprimedelista *)
```

Dos puntos hemos de explicar acerca del procedimiento *suprimedelista.* El primero se refiere a la depuración, o *debugging.* Al comienzo de este capítulo ya avisamos que la gente tiene problemas conceptuales para entender las estructuras de datos, tales como las listas; dicho de otro modo, sufre muchas equivocaciones. Tanto mayor, por consiguiente, es la importancia que tiene el diseñar los programas de modo que den buenos mensajes de error cuando las cosas van mal. Esa es la razón por la que hemos puesto la comprobación de errores al comienzo de nuestro procedimiento. Si la hubiéramos omitido, y se hubiese llamado a nuestro procedimiento con un argumento **nil**, el resultado habría sido (si Perkins estaba atento) un mensaje de error cuando se hizo referencia a *apuntdelista* ↑*.siguiente.* Inevitablemente, habría sido un mensaje de tipo general acerca del mal uso de los punteros, en lugar de uno específico y concreto, como el que nosotros podemos dar.

El segundo punto hacia el que queríamos llamar tu atención, es que has de ser muy cuidadoso para no hacer jamás referencia a nada de lo que te hayas deshecho por medio de *dispose.* Las tres últimas sentencias de nuestro procedimiento se podrían simplificar convirtiéndolas en:

```
dispose(apuntdelista);
apuntdelista: = apuntdelista ↑.siguiente;
```

pero esto está *mal,* porque *apuntdelista* ↑*.siguiente* queda dentro del avión recién eliminado con *dispose.* De hecho, en algunos compiladores el procedimiento *dispose* repone su argumento a **nil** para disuadir de tal trapacería.

Ahora mostraremos algunos ejemplos de llamada de *suprimedelista.* El ejemplo más sencillo es el de supresión de la cabeza de la lista,

```
suprimedelista(cabezadelista);
```

La figura 11.5 muestra la forma en que esto funciona en el caso en que los dos primeros elementos de la lista sean *AB*123 y *CD*456.

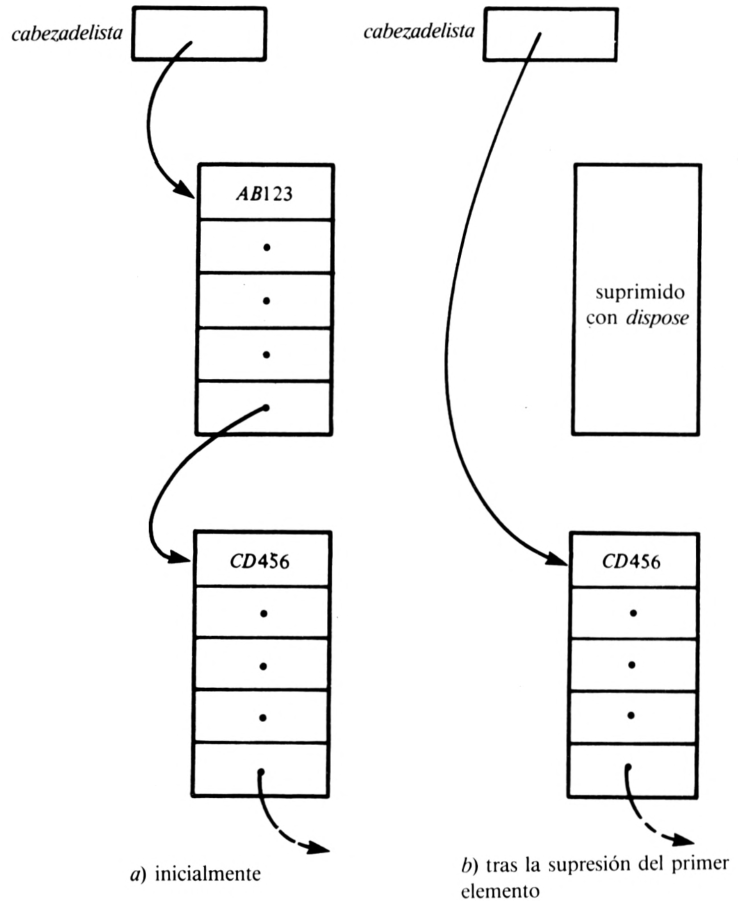


Figura 11.5. Supresión de la cabeza de una lista.

Lo que sigue borraría o suprimiría a todos los aviones de una lista:

```
while cabezadelista < > nil do
    suprimedelista(cabezadelista);
```

Finalmente, el siguiente código suprimiría el último elemento de una lista:

```
if cabezadelista < > nil then
begin (* sólo suprime de una lista que no sea nula *)
    if cabezadelista ↑.siguiente = nil then
        suprimedelista(cabezadelista) (* sólo 1 avión *)
    else
    begin
        (* Halla el penúltimo avión de la lista *)
```

```
        apuntactual: = cabezadelista;
        while apuntactual↑.siguiente ↑.siguiente < > nil do
            apuntactual: = apuntactual ↑.siguiente;
        suprimedelista(apuntactual ↑.siguiente);
    end;
end;
```

La aparente complicación adicional, haciendo un caso especial del primer elemento de la lista, tiene las mismas causas que ya vimos al agregar un elemento al final de una lista.

Para suprimir el último avión, tenemos que cambiar el campo *siguiente* del penúltimo avión. Por tanto, la estrategia del programa precedente es hallar el penúltimo avión y entonces llamar a *suprimedelista* para que opere sobre el campo *siguiente* de éste. Este campo *siguiente* apunta al avión que se ha de suprimir. El código contiene un ejemplar de puntero bastante bonito, a saber:

apuntactual ↑.*siguiente* ↑.*siguiente*

Para entenderlo, consideradlo como

(*apuntactual* ↑.*siguiente*)↑.*siguiente*

En otras palabras, tomamos el campo *siguiente* del *apuntactual* para obtener un puntero (el encerrado entre paréntesis), y luego usamos este puntero para extraer el campo *siguiente* de otro avión. Desde luego, se pueden construir ejemplares aún más bonitos usando, en vez de nuestros dos punteros, una secuencia de cuatro o cinco; pero recuerda que cada nivel de puntero duplica probablemente los problemas de cualquier lector que intente entender lo que estás haciendo.

También en este caso se puede escribir un procedimiento repetitivo que haga en forma mucho más limpia esta misma tarea, aunque la versión más larga que acabamos de ver contribuye a ilustrar algunos mecanismos. Otro punto para Primple.

Punteros adicionales

En algunas aplicaciones, las listas son largas y es bastante corriente que haya la necesidad de hallar el final de una lista. Cuando esto ocurre, nuestros bucles con **while**, que buscan los finales de listas, consumen demasiado tiempo; una alternativa es mantener una variable análoga a *cabezadelista,* pero que apunta al último avión de una lista. El mantenimiento de tal variable exige bastante *housekeeping* adicional. Pero, como ocurre con el *pudding* de carne y riñones de Heather, que lleva horas de preparación y proporciona sólo minu-

tos de delicia al comerlo, el esfuerzo puede valer la pena. En el caso de la optimización de un programa, uno invierte horas en el ajuste del programa, con el resultado de que el ordenador (así lo suponemos) saboree el exquisito placer de ejecutar un programa rápido.

Sumario sobre los usos del almacenamiento dinámico

Para muchos problemas, no necesitarás almacenamiento dinámico; las matrices o *arrays* normales, y cosas análogas, servirán muy bien a tus necesidades. Sin embargo, si empiezas a escribir programas para problemas más comprometidos, te darás cuenta, sea cual sea tu campo de aplicación, de que necesitas el almacenamiento dinámico. Tras un tiempo de dificultades para acostumbrarte a su compañía, comprobarás que los punteros son amigos verdaderos. Algunas operaciones, como la supresión de un elemento de una cabeza de lista, se hacen de verdad en forma mucho más fácil y eficiente usando el almacenamiento dinámico que con los *arrays* o matrices ordinarios.

Una biblioteca es «pensamiento» conservado en un frigorífico.

LORD SAMUEL

12 Bibliotecas

Este capítulo podría ser el más corto de todos los tiempos. Podría estar formado por esta única frase: "En PASCAL no existen bibliotecas". Sin embargo, vale la pena decir algo más.

Una de las grandes ventajas del PASCAL es la facilidad con que permite preparar procedimientos y funciones autónomos, o autocontenidos, que se pueden usar libremente en cualquier programa, sea tuyo o ajeno. Un buen programador debe, en cooperación con sus colegas, crear una biblioteca de tales programas y ponerla a la disposición de toda la comunidad de usuarios. La biblioteca podría almacenarse o conservarse en un disco.

Es bastante sorprendente, por tanto, que el informe PASCAL no hable de bibliotecas. Ahora trataremos de explicar por qué. En bien de la concisión, hablaremos sólo de procedimientos, pero lo que digamos es aplicable igualmente a las funciones de biblioteca.

Bibliotecas o códigos fuente y objeto

El primer punto que se ha de exponer es que las bibliotecas se pueden poner a disposición de los usuarios en forma fuente o en forma objeto. Como

hemos dicho anteriormente, un compilador convierte el programa que le entregamos, el *programa fuente,* en una forma binaria interna que los escritores de compiladores llaman *código objeto.* (Para esos compiladores que ejecutan los programas en forma muy lenta, ''código abyecto'' sería un nombre más adecuado.)

Si tienes un procedimiento de biblioteca largo, cuya compilación ocupe varios minutos, parece atractivo el convertirle en código objeto de una vez por todas, y poner a disposición de los usuarios de la biblioteca este código objeto, en lugar del programa fuente. En este caso, los programas de usuario se construyen con la ayuda de un programa del sistema al que algunas veces se llama un *enlazador* (*linker*). El enlazador une el programa objeto del usuario con el código objeto de cualesquiera procedimientos de biblioteca que necesite. Entonces, el programa resultante está listo para su ejecución.

Este método parece atractivo y, realmente, es muy utilizado. Es muy posible que lo hayas empleado en un sistema BASIC. Sin embargo, adolece de dos desventajas. La primera es que los programas enlazadores o *linker,* a pesar de su tarea aparentemente sencilla, son con mucha frecuencia lentos y complicados. Esta desventaja se potencia aún más como consecuencia de uno de los méritos del PASCAL: como este lenguaje ha sido diseñado teniendo presentes los problemas de la compilación, los compiladores son relativamente rápidos. (Desde luego, el tuyo puede ser una de las excepciones.) En consecuencia, muchas veces es más rápido compilar de nuevo un procedimiento que incurrir en la pérdida de tiempo adicional que supone el aplicar el enlazador a su código objeto.

La segunda desventaja que tiene el enlace con procedimientos de código objeto es más fundamental: desconcertamos y molestamos a Perkins. Una de las aportaciones de Perkins al PASCAL es que se ocupa de verificar, en todas las llamadas que se hagan a procedimientos, que los argumentos y los parámetros concuerden en cuanto a número, tipo de datos, etc. Lo hace de una vez para siempre en el momento de la compilación, de modo que en el de la ejecución tengas una cómoda sensación de seguridad, pero sin ninguna carga adicional de comprobaciones. Toda esta estrategia se basa en la declaración de los procedimientos en el programa que los utiliza, de modo que Perkins puede echar un vistazo al encabezamiento de los mismos y compararle con las llamadas. Si existe una posibilidad de compilar por separado los procedimienos y el programa que los usa, puede perderse la seguridad.

Tal vez no sea obvia inmediatamente la razón para esta falta de seguridad. La mayor parte de los sistemas PASCAL que contienen ampliaciones o extensiones para soportar bibliotecas de código objeto, exigen, en efecto, que el programa que utilice la biblioteca —el programa *llamador*— declare un encabezamiento por cada procedimiento de la biblioteca utilizado. En lugar del cuerpo de tal procedimiento, se escribe la palabra *externo* (o análoga). Así, si un programa *llamador* hace uso del procedimiento de biblioteca *útil,* que toma un parámetro real, necesitaría contener la declaración

procedure *útil*(*x*: *real*); *extern*;

Entonces, efectivamente, Perkins podrá comprobar que todas las llamadas a *útil* tengan los argumentos apropiados.

La falta de seguridad surge porque el procedimiento contenido efectivamente en la biblioteca puede no coincidir con su descripción dada en el programa llamador. El verdadero procedimiento *útil* puede, por ejemplo, tomar un parámetro matricial. El resultado es que el programa se hace un lío. Si tienes suerte, producirá rápidamente un mensaje de error; si no la tienes, puede volverse loco y corromper todo tu sistema.

Ahora bien, si deliberadamente has declarado que un procedimiento es algo distinto de lo que es en realidad, con la finalidad de hacer trucos como los que haces en los registros con las uniones indiscriminadas, te merecerás lo que te pase. Sin embargo, es fácil declarar mal un procedimiento en forma completamente accidental, y en este caso el castigo puede ser mucho mayor que el delito.

Bibliotecas de códigos objeto con seguridad

Algunos sistemas PASCAL realizan la hazaña de dotar de una seguridad completa a las bibliotecas de códigos objeto. Para ello, hay que agrupar entre sí, formando una unidad:

- La declaración que necesita un programa llamador.
- El código objeto del procedimiento.

Ambas partes se crean automáticamente a partir del código fuente del procedimiento, y el programa llamador no puede acceder a una de ellas sin la otra. Como mecanismo de un tipo más general, algunos sistemas realmente excelentes permiten agrupar en una biblioteca cualquier clase de declaraciones. Esto podría incluir, además de las declaraciones de procedimientos, las de constantes y de tipos. Para almacenar en una biblioteca abstracciones tales como las utilidades para números complejos que hemos visto en el capítulo 8, es necesario este tipo de facilidad.

Los sistemas PASCAL de esta naturaleza son normalmente los que funcionan con sistemas operativos diseñados pensando en el PASCAL. Aquellos PASCAL que tienen que lidiar con sistemas operativos poco cooperativos tienen muchas más dificultades para ofrecer seguridad.

Si tienes un sistema PASCAL que ofrezca bibliotecas seguras de código objeto, considérate extremadamente afortunado: sean cuales sean los demás problemas de tu vida, gozas de la bendición de un don auténtico.

Bibliotecas en otros lenguajes

Puede ocurrir que, dentro de tu programa PASCAL, desees usar procedimientos escritos en otros lenguajes. La necesidad de ello puede surgir por dos causas. Primera, puedes tener un procedimiento existente que funcione bien, y no deseas tomarte el trabajo de traducirlo al PASCAL. Segunda, puede ocurrir que desees realizar alguna operación que en PASCAL sea imposible. Entonces, la única forma de salvar el inconveniente es escribir en algún otro lenguaje, por ejemplo en ensamblador, un procedimiento para conseguir el efecto deseado, y utilizar este procedimiento en tu programa PASCAL.

Al llamar a procedimientos en lenguajes extraños, es frecuente que haya problemas para la representación de los tipos de datos. Puede ocurrir, por ejemplo, que los distintos lenguajes almacenen los *arrays* o matrices en formas diferentes. Por consiguiente, los problemas de seguridad son inmensos, y el mejor consejo es que, mientras sea posible, se evite el uso de procedimientos extraños al lenguaje.

A pesar de este consejo, Bill mostró un interés poco usual en el tema. Averiguamos con gusto que había llegado incluso a escribir un programa en PASCAL, aunque cuando vimos el programa nos sentimos bastante menos impresionados. Era el siguiente:

```
program x(input, output);
procedure miprogramabasic; extern;
begin
    miprogramabasic;
end.
```

No obstante, era un esfuerzo que indicaba voluntad, y no tuvimos valor para decirle que son muy pocos los sistemas que permiten la llamada de programas BASIC desde dentro de programas PASCAL.

La sentencia CHAIN

En el PASCAL estándar no existe equivalente alguno de la sentencia CHAIN del BASIC. Sin embargo, al igual que ocurre con las bibliotecas en código objeto, algunos compiladores PASCAL ofrecen facilidades extra de esta naturaleza, con seguridad o sin ella.

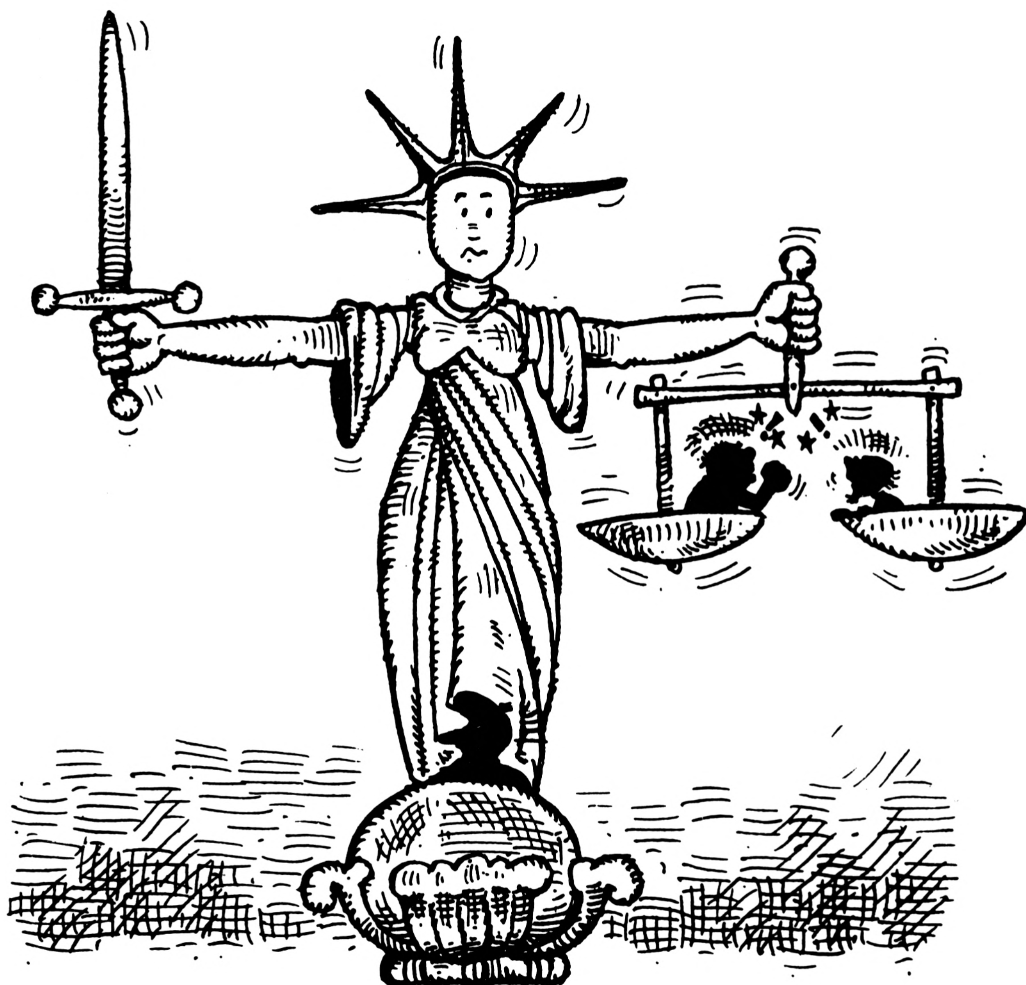
Bibliotecas en código fuente

Supongamos que, por las razones que sean, has decidido construir tu biblioteca en código fuente, y no en código objeto. Es indudable que no puede haber objeciones prácticas ni morales al uso de bibliotecas en código fuente. Verdaderamente, no las hay; pero, sin embargo, hay todavía muchos compiladores PASCAL que no soportan esas bibliotecas. La razón es que, para mantenerse dentro de un tamaño razonable, el PASCAL se esfuerza al máximo en no asumir las funciones de otros programas o elementos de *software,* tales como el sistema operativo o el editor. La inclusión de bibliotecas está considerada como una tarea de este *software* externo.

Desgraciadamente, éste no es usualmente un enfoque ideal, y es mejor disponer dentro del programa PASCAL de una orden o comando para incluir en él el material procedente de bibliotecas en código fuente. Por esto es por lo que muchos PASCALES soportan una orden o comando tal como

include filename;

Si tu compilador dispone de una facilidad *include,* debes aprovecharla al máximo. Aunque se trata de una característica no estándar, no pone en peligro la portabilidad. Si necesitas trasladar tu programa a otro ordenador, siempre podrás construir un programa completo utilizando todos sus trozos o fragmentos introducidos con *include.* Es más, y para decirlo en la forma más vigorosa: si tienes una buena facilidad *include,* en contraposición con una facilidad poco segura para bibliotecas en código objeto, olvídate de esta última; sólo cuando su necesidad se haga realmente apremiante, deberás descender al nivel del código objeto.



No hay nada como una comida gratis.

El aforismo más verdadero en la informática

13 Resumiendo

El saldo

Vamos a terminar con un breve resumen de lo que ganas y lo que pierdes al pasarte al PASCAL.

La ganancia más importante, como hemos destacado todo el tiempo, es la legibilidad. Si un gran programa no es legible, no es nada.

El profesor Primple nunca escribe comentario alguno en sus programas en PASCAL. "El PASCAL es autoexplicativo y se documenta a sí mismo", enuncia. Sin embargo, las mentes no tan privilegiadas hallan sus programas imposibles de entender, y encontrarían más fácil un programa BASIC bien comentado. Así pues, si tu programa ha de ser leído por estos mortales menos eminentes, no desprecies las ventajas de la legibilidad fundamental del PASCAL omitiendo los comentarios. Necesitas menos comentarios de bajo nivel que en BASIC ("Esto es un bucle"), pero todavía necesitarás explicar la estrategia de nivel más alto de tu programación.

Cuando se consigue la legibilidad, siempre viene seguida de otras innúmeras ventajas en el terreno de la corrección de los programas y de su mantenimiento.

Otros beneficios que produce el PASCAL son los relativos a la portabilidad, la seguridad y la estructuración.

El principal precio que hay que pagar por todo ello es tu inversión inicial de tiempo y paciencia. Ya has hecho un pago parcial, tal vez bastante grande, al haber avanzado laboriosamente hasta el final de este libro. Sin embargo, el "leer" es sólo una pequeña parte en el aprendizaje de un nuevo lenguaje; el "hacer" es una parte mucho más importante.

Cuando se entra en contacto con un elemento de *software* nuevo, como cuando se conoce por primera vez a una persona, es poco probable que se produzca una amistad inmediata con él. Inicialmente, tu nuevo conocido te parecerá raro y caprichoso, y te será difícil establecer la comunicación. La prueba principal de su valor llega después de varias semanas en su compañía. En algunos casos, para entonces habrás llegado a la conclusión de que es aún más raro y caprichoso de lo que habías pensado al principio, y de que la comunicación con él es todavía más difícil, en cuyo caso será el momento de decirle adiós. Otras veces, en cambio, comprobarás que, bajo un aspecto exterior antipático o poco atractivo, se ocultaba lo que ahora ha llegado a ser un amigo cordial.

Al hacer el cambio al PASCAL, conoces nuevos editores, otros sistemas operativos, compiladores y depuradores o *debuggers*; y el tratar de hacer amistad con todos ellos es una considerable tarea, sean cuales sean los méritos del lenguaje PASCAL. A menudo resulta particularmente difícil asimilar un nuevo estilo para la depuración de programas.

Al mismo tiempo que te vas acostumbrando a estos programas de sistemas, tienes que ir creando una comprensión más íntima del PASCAL propiamente dicho.

La adaptación del estilo

A pesar de su fracaso con el programa de la biblioteca, Bill Mudd ha escrito un segundo programa PASCAL. Es como sigue:

```
program CUENTAS(INPUT,OUTPUT);
(* REM COPYRIGHT BILL MUDD, 1982 *)

label
10,20,30,40,50,60,70,80,90,100,110,120,130,140,
200,210,220,230,240,250,999;
var
A,B,C,D,E,F,G,H,I,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z: REAL;
J,K,L:INTEGER; (* REM PARA USAR EN SENTENCIAS FOR *)
ADOLAR, BDOLAR, CDOLAR, DDOLAR:CHAR;
```

```
(* REM ESTOS PROCEDIMIENTOS EVITAN LA CONFUSION DE
USAR READ PARA INPUT *)
procedure INPUTN(var X:REAL);begin;read(X);end;
procedure INPUTS(var X:CHAR);begin;read(X);end;
begin;
(* REM YO SIEMPRE PONGO ESTAS DECLARACIONES ANTE-
RIORES AL PRINCIPIO DE TODOS MIS PROGRAMAS EN PAS-
CAL. PUEDO VENDER ESTA VALIOSISIMA AYUDA A QUIEN LA
DESEE POR SOLO CIEN LIBRAS *)
10:   INPUTN(X);
20:   if X>0 then goto 50;
30:   WRITELN('TECLEE UN NUMERO POSITIVO');
40:   goto 10;
50:   S:=0;
60:   for K:=1 to TRUNC(X) do begin;
70:   (* REM LEE X NUMEROS Y LOS SUMA *);
80:   INPUTN(Q);
90:   if Q< >99.99 then goto 130;
100:  (* REM 99.99 SIGNIFICA DATO FINAL. PASA K A VALOR
      ALTO PARA PARAR EL BUCLE *);
110:  K:=12345;
120:  goto 140;
130:  S:=S+Q;
140:  (* NEXT K *)end;
200:  WRITELN(S/X, 'ES LA MEDIA DE LOS NUMEROS');
210:  WRITELN('¿DESEA USTED CONTINUAR?');
220:  INPUTS(ADOLAR);
230:  if ADOLAR='N' then goto 999;
240:  if ADOLAR< >'Y' then goto 210;
250:  goto 10;
999:  HALT;
end.
```

No sólo ha escrito un programa PASCAL, sino que ha tenido la amabilidad de pasarnos el fruto de su experiencia: ''Si tenéis que programar en PASCAL, lo que os aconsejo es que pongáis *labels* o etiquetas para todas las líneas, de modo que, si más adelante os parece bien, siempre podáis hacer GOTO a cualquiera de ellas. También sirve de ayuda al editar. He escrito un editor especial (en BASIC, desde luego) que agrega automáticamente las *labels* o etiquetas, y también pone el punto y coma al final de cada línea. ¿Sabéis lo que os digo? Si uno se esfuerza, puede llegar a convertir al PASCAL en un lenguaje bastante majo.''

''Al principio tuve algunas dificultades con el Perkins ese'', continuó. ''Se oponía a que yo cambiara el valor de K en la línea 110. Sin embargo, hace poco he tenido la suerte de hallar un compilador que casi no hace ninguna comprobación. Con él se puede alcanzar pronto la fase de ejecución del programa,

y entonces se puede poner manos a la tarea de conseguir que las respuestas salgan bien.''

Es estupendo ver a Bill realizar un esfuerzo tan grande, pero, desgraciadamente, si vas a escribir los programas PASCAL como los BASIC, harías mucho mejor en continuar con éste. Para escribir verdaderos programas PASCAL, hay que hacer el esfuerzo preciso para dominar los conceptos nuevos. Entre los que más hemos destacado, figuran:

- las sentencias de formación de bucles;
- la estructura en bloques y las variables locales;
- la selección de los tipos de datos de manera que el programa se adecúe al problema;
- el almacenamiento dinámico.

El dominar todo esto te llevará algún tiempo, pero el aprendizaje de conceptos nuevos y claros es realmente divertido, de modo que el tiempo pasará rápidamente. El tiempo empleado en dominar conceptos que son más familiares, pero menos claros, como el de *input/output* en PASCAL, te parecerá más largo.

El esfuerzo en la preparación de programas

Incluso después de haber llegado a dominar el PASCAL, te darás cuenta de que las primeras fases de la preparación de los programas te llevan más tiempo que con el BASIC. Tienes el trabajo adicional de declararlo todo, y, en comparación con el BASIC, empleas mucho más tiempo en la localización y la corrección de los errores de sintaxis. De vez en cuando te sentirás obstaculizado, al impedirte el PASCAL que hagas algo que realmente desees hacer. Estas molestias, sin embargo, no constituyen realmente un coste debido al empleo del PASCAL; el tiempo adicional que se invierta inicialmente en la preparación de un programa disciplinado, se verá más que adecuadamente compensado después si el programa tiene cierta magnitud.

Si tu programa es pequeño y de fácil compresión, no dudes en seguir utilizando el BASIC. Muchos programadores experimentados de PASCAL utilizan el BASIC para los programas pequeños, especialmente para aquellos que tratan de manipulación de cadenas o *strings* y de entrada/salida; sin embargo, jamás lo confesarían ante sus colegas.

La ampliación o extensión

Si deseas hallar más información acerca del lenguaje PASCAL y de la forma en que está implementado, un excelente punto de partida es la colección de artículos titulada *PASCAL. El lenguaje y su implementación,* editada por D. W. Barron (1981). Este trabajo suscita la interesante idea de que un lenguaje sea "mejor que sus sucesores". Ciertamente, existe una importante corriente de opinión según la cual, si un lenguaje está bien construido y todas sus partes encajan armoniosamente, los intentos de ampliarle o de restringirle no tienen más probabilidades de éxito que las que podría tener una nueva versión abreviada (o ampliada) de una sinfonía de Beethoven.

Se han hecho muchas tentativas consistentes en tomar unas cuantas características del PASCAL y pegárselas al BASIC. Es posible que el híbrido resultante sea un lenguaje utilizable, pero decir que ofrece todas las ventajas del PASCAL sería pura charlatanería.

Palabras finales

Bill insiste en decir él la última palabra y, al parecer, ha abandonado aquellos esfuerzos en el PASCAL que hemos descrito tan recientemente.

> "He probado el PASCAL", dice, "pero me di cuenta de que estaba invirtiendo demasiado tiempo en aprender el lenguaje y en conseguir que el compilador aceptara mis programas. Yo soy un hombre muy ocupado. Necesito todo este mes para conseguir que funcione el programa BASIC de mil líneas con el que estoy actualmente, y pienso emplear el mes que viene tratando de averiguar cómo funciona un gran programa BASIC que escribí el invierno pasado. Así que, desde luego, no dispondré de ningún tiempo de sobra para malgastarlo en el PASCAL."

Referencias

Atkinson, L. V. y North, S. D. (1981). COPAS. Un sistema PASCAL conversacional, *Software. Práctica y Experiencia* **11**, 8, pp. 819-30.

Barron, D. W. (editor), (1981). *PASCAL. El lenguaje y su implementación.* Wiley, Chichester.

Bishop, J. M. (1979). Implementación de cadenas o *strings* en PASCAL, *Software. Práctica y Experiencia* **9**, 9, pp. 711-18.

Grogono, P. (1980). *Programando en PASCAL.* Addison-Wesley, Reading, Mass.

Jensen, K. y Wirth, N. (1975). *Informe y manual del usuario de PASCAL.* Springer-Verlag, Nueva York.

Kaye, D. R. (1980). Introducción (*input*) interactiva en PASCAL, *Noticias SIGPLAN* **15**, 1, pp. 66-9.

Kemeny, J. G. y Kurtz, T. E. (1980). *Programación en BASIC,* tercera edición. Wiley, Nueva York.

Kernighan, B. W. y Plaugher P. J. (1981). *Las herramientas de software en PASCAL.* Addison-Wesley, Reading, Mass.

Knuth, D. E. (1973). *El arte de la programación de ordenadores.* Addison-Wesley, Reading, Mass.

Ledgard, H. F., Hueras, J. F. y Nagin, P. A. (1979). *El PASCAL con estilo.* Hayden, Rochelle Park, N. J.

Nevison, J. M. (1978). *El librito de estilo en BASIC.* Addison-Wesley, Reading. Mass.

Parnas, D. L. (1972). Sobre los criterios a utilizar para la descomposición de sistemas en módulos, *Comm. ACM* **15**, 12, pp. 1053-8.

Richards, M. y Whitby-Strevens, C. (1979). *BCPL. El lenguaje y su compilador.* Cambridge University Press.

Sale, A. (1979). Implementación de cadenas en PASCAL, una vez más, *Software. Práctica y Experiencia* **9**, 10, pp. 839-42.

Standish, T. A. (1980). *Técnicas de estructuras de datos.* Addison-Wesley, Reading, Mass.

Teitelbaum, T. y Reps, T. (1981). El sintetizador de programas de Cornell: Un entorno de programación orientado a la sintaxis, *Comm. ACM* **24**, 9, pp. 563-73.

Teitelman, W. (1977). Un auxiliar del programador orientado a la presentación visual, o *display, Documentos de la V Conferencia Internacional Conjunta sobre Inteligencia Artificial* **2**, pp. 905-15.

Turner, D. A. (1979). Una nueva técnica de implementación para lenguajes interpretativos, *Software. Práctica y Experiencia* **9**, 1, pp. 31-50.

Welsh, J., Sneeringer, W. J. y Hoare, C. A. R. (1981). Ambigüedades e inseguridades en PASCAL, véase Barron (editor), pp. 5-20.

Wirth, N. (1971). El desarrollo de programas por medio del refino progresivo, *Comm. ACM* **14**, 4, pp. 221-7.

Wirth, N. (1976). *Algoritmos + Estructuras de datos = Programas.* Prentice-Hall, Englewood Cliffs, N. J.

Apéndice A: Procedimientos y funciones incorporados

En este libro se han mencionado la mayoría de los procedimientos y funciones incorporados en el PASCAL. Aquí los presentamos todos reunidos en un mismo sitio. Es indudable que cada implementación tendrá algunas adiciones a nuestra lista, pero nosotros nos limitamos aquí a las rutinas definidas en el informe del PASCAL.

Las letras que empleamos para representar argumentos indican el tipo de datos de los mismos en la forma siguiente:

f significa archivo (*file*),
i significa entero (*integer*),
p significa puntero (*pointer*),
r significa real,
s significa un tipo definido por el usuario, o *Boolean* o *char*.

Procedimientos

A continuación se da una lista de los procedimientos incorporados:

a) Concernientes a ENTRADA/SALIDA. Véase el capítulo 9

get(*f*)	avanza en un componente la exploración del archivo *f* y asigna el valor del nuevo componente a *f*↑;
page(*f*)	inicia una nueva página en el archivo de texto de salida *f*;
put(*f*)	agrega o pospone *f*↑ al archivo *f*;
reset(*f*)	prepara el archivo *f* para su lectura, actualizando *f*↑ al valor del primer componente;
rewrite(*f*)	prepara el archivo *f* para la escritura, destruyendo cualquier versión anterior del mismo.

Además existen los procedimientos *read, readln, write* y *writeln,* que toman un número variable de argumentos.

b) Concernientes a la asignación de espacio de almacenamiento o memoria. Véase el capítulo 11

dispose(*p*)	devuelve el espacio de memoria ocupado por la variable dinámica a la que apunta *p*;
new(*p*)	toma memoria prestada para una variable dinámica y establece a *p* para que apunte a ella.

c) Concernientes a las matrices compactadas. Véase el capítulo 7

pack(*au, índice, ap*)	compacta la matriz *au* dentro del *ap,* empezando con el elemento *au*[*índice*];
unpack(*ap, au, índice*)	inverso del *pack*.

Aquí *ap* es una matriz compactada, *au* es una no compactada del mismo tipo, e *índice* es un índice aceptable para estas matrices.

Funciones

A continuación se da una lista de funciones incorporadas. Las concernientes a la aritmética se presentaron en el capítulo 4, las relativas a entrada/salida, en el 9, y *ord, pred* y *succ,* en el 6. Las funciones *odd* y *round* se describen por primera vez aquí.

a) Que producen un resultado entero (*integer*)

abs(*i*)	el valor absoluto de *i*;
ord(*s*)	el número ordinal de *s* en el conjunto de valores posibles de *s*;
round(*r*)	el entero más próximo a *r*;
sqr(*i*)	el cuadrado de *i*;
trunc(*r*)	el entero que se obtiene suprimiendo la parte fraccionaria de *r*.

b) Que producen un resultado Booleano

eof(*f*)	*true* si *f* está situado en el final del archivo;
eoln(*f*)	*true* si *f* está situado al final de la línea;
odd(*i*)	*true* si *i* es impar.

c) Que producen un resultado real

abs(*r*)	el valor absoluto de *r*;
arctan(*r*)	el arcotangente de *r*;
cos(*r*)	el coseno de *r*;
exp(*r*)	*e* elevado a la potencia *r*;
ln(*r*)	el logaritmo natural de *r*;
sin(*r*)	el seno de *r*;
sqr(*r*)	el cuadrado de *r*;
sqrt(*r*)	la raíz cuadrada de *r*;

d) Que producen como resultado un carácter

chr(*i*)	el carácter cuyo número ordinal es *i*.

e) Que producen el mismo tipo del argumento

pred(*s*)	el predecesor de *s*;
succ(*s*)	el sucesor de *s*.

Las funciones trigonométricas (*arctan, cos, sin*) funcionan en radianes.

Apéndice B: Sumario del PASCAL

El objeto de este Apéndice consiste en presentar un programa completo que ilustre la mayoría de las características del PASCAL. Es un complemento al Apéndice C, que contiene una especificación sintáctica mucho más precisa.

Si estás escribiendo programas en PASCAL, este Apéndice te podrá ser útil en dos formas. En primer lugar, te podrá ser útil como consulta o referencia rápida si se te han olvidado algunos detalles sintácticos. En segundo lugar, si repasas el programa línea por línea, te podrá recordar alguna característica potente del PASCAL que no hayas estado usando. Sin embargo, no trates de averiguar lo que hace este programa: no hace nada sensato en absoluto; es simplemente un catálogo de ejemplos aislados.

Para mantener este ejemplo de programa razonablemente corto, hemos omitido algunos de los conceptos más caprichosos o lujosos, como el **with**, los registros con variantes y los archivos binarios.

```
(* ************************
    Encabezamiento del programa
    ************************ *)

program xxx(input, output);
(* o, si utiliza archivos externos
```

```
program xxx(f1,f2,input,output);
*)

(* ***********************************************
   Declaraciones de etiquetas o labels, constantes, tipos y variables
   *********************************************** *)

label
   999,15;

const
   tamañogrupo = 20;
   peculiar = 'x';
   pi = 3.14159;

type
   quesos = (Stilton, Cheddar, Cheshire, Wensleydale); (* tipo definido
                                                          por el usuario *)
   porcentaje = 0..100; (* un tipo de subrango *)
   tablaquesos = set of quesos;
   array1 = array [1..tamañogrupo, quesos] of real; (* el segundo sub-
                                                       índice es un queso *)
   array2 = packed array [1..6] of char; (* una cadena o string *)
   array3 = array [porcentaje] of Boolean;
   petición =
      record
         puntuación: porcentaje;
         correcto: Boolean;
      end;
   listapunt = ↑colección; (* un puntero *)
   colección =
      record
         campo1: real;
         campo2: integer;
         campoarray: array [1..10] of quesos;
         siguiente: listapunt
      end;
var
   índice, vint: integer;
   vreal1, vreal2: real;
   estálloviendo: Boolean; (* puede tomar los valores true o false *)
   miinicial: char; (* un solo carácter *)
   mejorqueso: quesos;
   mipuntuación: porcentaje;
   mielección, suelección: tablaquesos;
   a1: array1;
   respuesta: array2;
   ocurre: array3;
```

```
    apuntdecabeza: listapunt;
    resultexamen: petición;
    f1, f2: text; (* archivos de texto *)

(* **********************************
    Declaraciones de funciones y procedimientos
    ********************************** *)

    function (param1: integer; param2: quesos): real; (* El resultado de la
                                                         función es real *)
    var
        milocal1: quesos;
        milocal2: integer;
    begin
        milocal2: = param1 div 3;
        for milocal1: = Stilton to param2 do
            milocal2: = param1 * (milocal2 + 1);
        f: = milocal2/6; (* asigna el resultado de la función *)
    end; (* f *)

    procedure puestacero (var x: integer);
    (* Este procedimiento pone simplemente a su parámetro en cero.
       No tiene ninguna declaración local *)
    begin
        x: = 0;
    end; (* puestacero *)

    begin(* programa principal *)
    (* ************************************
        Ejemplos de constantes, expresiones y sentencias
        ************************************ *)

    (* 1: asignaciones en las que se muestran constantes y expresiones *)

          vreal1: = 3.577E-2; (* una constante real *)
          vint: = vint mod 3 — vint div tamañogrupo; (* operadores espe-
                                                       ciales de integer
                                                       *)
          estálloviendo: = true;
          miinicial: = 'p';
          mejorqueso: = Stilton;
          suselección: = [Cheshire]; (* un constructor de conjuntos *)
          miselección: = [Stilton..Wensleydale];
          suselección: = suselección + [mejorqueso]; (* una unión de con-
                                                       juntos *)
          apuntdecabeza: = nil;
```

```
(* 2: condicionales *)

      if estálloviendo then
         vint:= vint+ 1;
      if (apuntdecabeza= nil) or not (Cheddar in miselección) then
      begin (* una sentencia compuesta *)
        vreal2: =cos(vreal1);
        if vreal1 > vreal2 then (* un if anidado *)
            vreal1: =vreal1 + 1;
      end else
        vreal1: = -vreal2;
      case miinicial of
        'p':
            writeln('¿Te llamas Pedro?');
        'w','b':
            writeln('¿Te llamas William o Bill?');
        peculiar:
            begin
                writeln ('No conozco ningún nombre que empiece así.');
                writeln ('¿Estás seguro de que está bien?');
            end;
      end;

(* 3: llamadas de procedimientos y de funciones *)

      puestacero(vint);
      vreal1: =f(6, mejorqueso)/3.9;

(* 4: referencias a arrays, registros y punteros *)

      a1[6, Cheddar]: = pi;
      respuesta: = 'abcdef'; (* constante de cadena o string *)
      ocurre [mipuntuación]: =false;
      resultexamen.puntuación: = 100;
      resultexamen.correct: = (resultexamen.puntuación < 62)and ocurre
                                    [resultexamen.puntuación];
      new(apuntdecabeza);
      apuntdecabeza↑.campo1: = 6;
      apuntdecabeza↑.campodearray[4]: =Stilton;
      dispose(apuntdecabeza);

(* 5: bucles *)

      for índice: = 1 to tamañogrupo do
        a1[índice, Cheddar]: = 0.12;
      for índice: = vint + 4 downto 2 do
        respuesta[índice + 1]: = respuesta[índice];
      for mejorqueso: = Stilton to Wensleydale do
        a1[vint + 1, mejorqueso]: =1E5;
```

```
índice:= 10; (* frecuentemente, un while va precedido de iniciali-
                zación *)
while ocurre[índice] and (índice < 40) do
  índice:= índice + 2;
mejorqueso:= Stilton; (* un repeat va frecuentemente precedido
                         de inicialización *)
repeat
  mejorqueso:= succ(mejorqueso);
  mipuntuación: mipuntuación + 1;
until mejorqueso in suselección * miselección + [Wensleydale];

(* 6: entrada/salida *)

    read(mipuntuación);
    readln(mipuntuación, vreal1, vreal2);
    write('La respuesta es');
    writeln(respuesta, 'y mipuntuación es', mipuntuación:3);
    reset(f1);
    rewrite(f2);
    read(f1, vreal1); (* entrada desde un archivo *)
    writeln(f2, respuesta, estálloviendo); (* salida a un archivo *)

(* 7: sentencias goto *)

    (* Yo he suprimido esta parte.
            Profesor Marcus d'A. Primple, Ph.d. *)

(* Estas son las etiquetas o labels que no han llegado a usarse *)

999:
15:
end. (* xxx *)
(*  PS
        goto 999;
            Bill Mudd, B.Sc.(suspendido) *)
```

Apéndice C: Diagramas de sintaxis

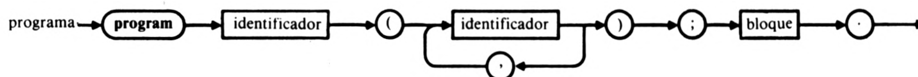
Explicación

Los diagramas de sintaxis constituyen una forma gráfica muy cómoda y conveniente para definir la sintaxis precisa de un lenguaje de programación. Este Apéndice contiene diagramas de sintaxis para el PASCAL basados en los diagramas de Grogono (1980)*. Por cierto, es posible que desees consultar este libro de Grogono. Da una descripción del PASCAL en un nivel mucho más detallado que el de éste.

La regla para la utilización de un diagrama de sintaxis es sencilla: si es posible formar cierto conjunto de símbolos siguiendo las flechas del diagrama, ese conjunto de símbolos será PASCAL sintácticamente correcto; si no se puede formar, no lo es. Los símbolos contenidos en las cajas o recuadros especifican los elementos constituyentes del programa. Los nombres que aparecen en minúsculas, pero no negritas, como ''bloque'' o ''identificador'' —siempre aparecen en cajas rectangulares—, se refieren a elementos constituyentes definidos

* Grogono, *Programming in Pascal,* © 1978, Addison - Wesley, Reading, Massachusetts, pp. 324-33.

en otros diagramas de sintaxis; todo lo demás, por ejemplo "**program**" o ";", especifica símbolos que se pueden usar en la construcción de un programa. Tomaremos como ejemplo el primer diagrama de sintaxis. Se reproduce a continuación para mayor comodidad.



El nombre que hay a la izquierda del diagrama nos dice lo que éste define. Así, el diagrama que aparece más arriba es el más fundamental de todos, porque define lo que es un "programa". Tomando este diagrama y siguiendo las flechas desde el principio, vemos que un programa consta del símbolo **program** seguido por un identificador, seguido por un "(" y otro identificador. Entonces podemos tomar cualquiera de dos ramas. Una consiste en seguir recto, pasando por ")", ";", "bloque" y "." Esto nos dice que un posible programa es

program *<identificador>* (*<identificador>*); *<bloque>*.

en el que la notación <identificador> significa cualquier cosa que encaje en el diagrama de sintaxis "identificador", y lo mismo se puede decir de <bloque>.

Si, en lugar de seguir la rama recta o directa, realizamos un bucle hacia atrás, terminamos con

program *<identificador>* (*<identificador>*), *<identificador>*

En este punto, nos encontramos de nuevo con la misma decisión. Podemos hacer otro bucle hacia atrás e incluir otra coma y otro identificador, o seguir rectos hasta el final. El significado general es que podemos tener, dentro de los paréntesis, tantos identificadores como queramos, separados por comas.

La ventaja de esta notación es que, a diferencia de las explicaciones verbales dadas en este libro, es absolutamente precisa. Sabes, por ejemplo, que ha de haber por lo menos un identificador dentro de los paréntesis. No se puede escribir

program *xxx*;

sin paréntesis alguno. Por otra parte, si miras la definición de una lista de parámetros para un procedimiento —ve el diagrama de sintaxis titulado "lista de parámetros"—, verás que en este caso se puede omitir la lista entre paréntesis (siguiendo la línea superior de derivación).

Una vez que hemos hallado la sintaxis general de un programa, tenemos que rellenar las definiciones de "identificador" y de "bloque". La definición de "bloque" es especialmente interesante, y puede dar respuesta a muchas preguntas. Por ejemplo, la respuesta a "¿Es posible omitir la sección **var**?" es

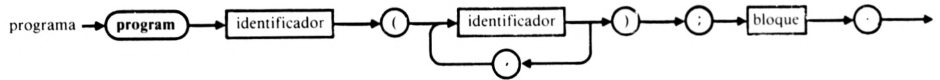
afirmativa, pero la de la pregunta "¿Puede haber dos secciones **var**?" no lo es. (Todas las líneas son como calles de sentido único, y está prohibido el giro de 180 grados.)

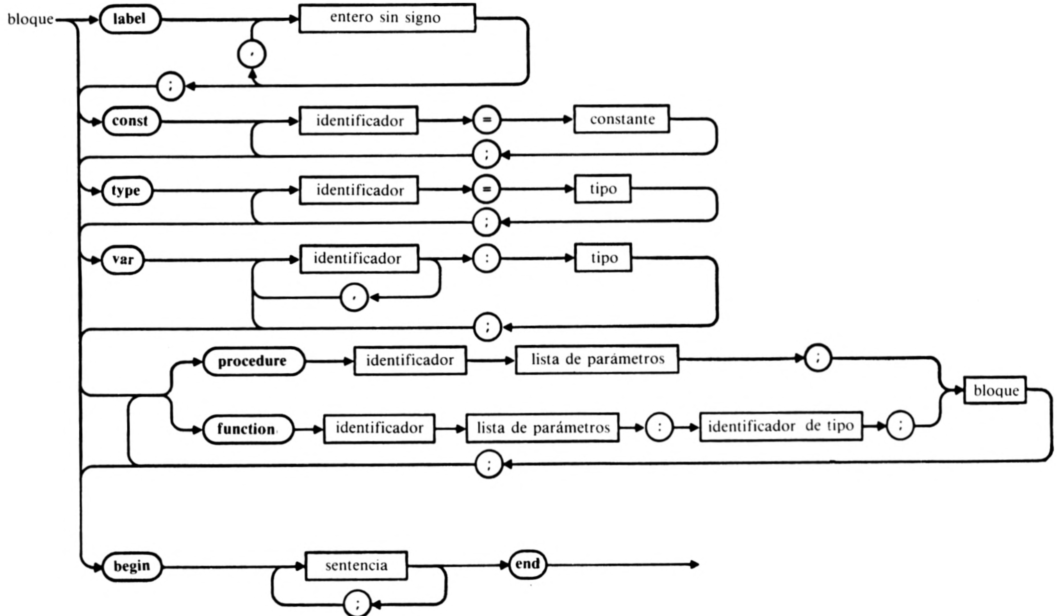
Finalmente, si vamos hacia el final de los diagramas de sintaxis podemos encontrar que un indentificador es una letra seguida por un número arbitrario de letras o cifras. Los dos últimos diagramas definen lo que son letras y cifras, pero encierran pocas sorpresas. (De hecho, la mayoría de los Pascales soportan letras minúsculas, además de las mayúsculas.)

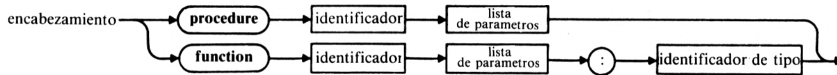
Limitaciones

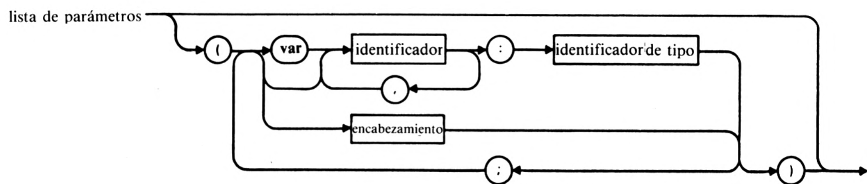
Obsérvese que estos diagramas, aunque tienen un enorme valor, no lo definen todo. No dicen nada acerca del espaciado, los cambios de línea ni los comentarios, ni especifican la regla según la cual tenemos que declarar todos los identificadores que usemos. Son cuestiones que no se prestan bien a la notación gráfica, y es necesario definirlas en alguna otra forma. Normalmente, se definen en forma verbal, como lo hemos hecho nosotros en este libro.

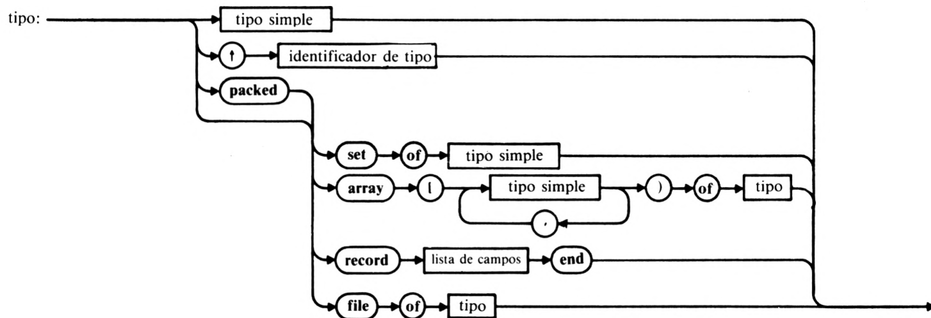
Diagramas de sintaxis del PASCAL estándar

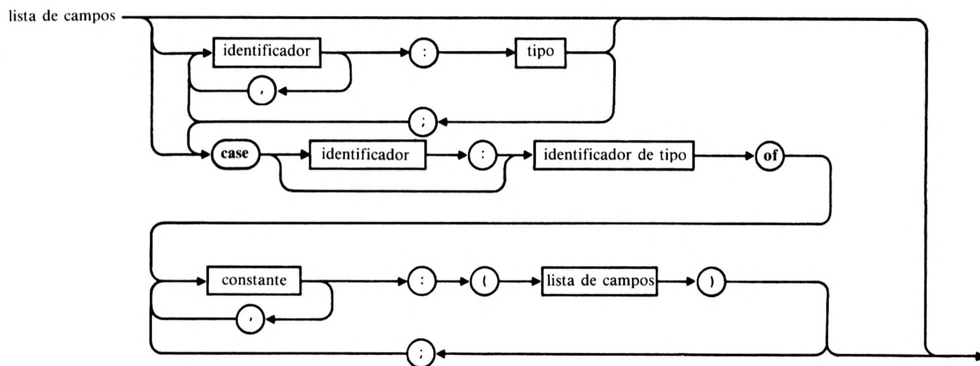


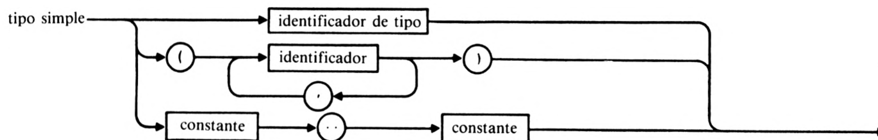


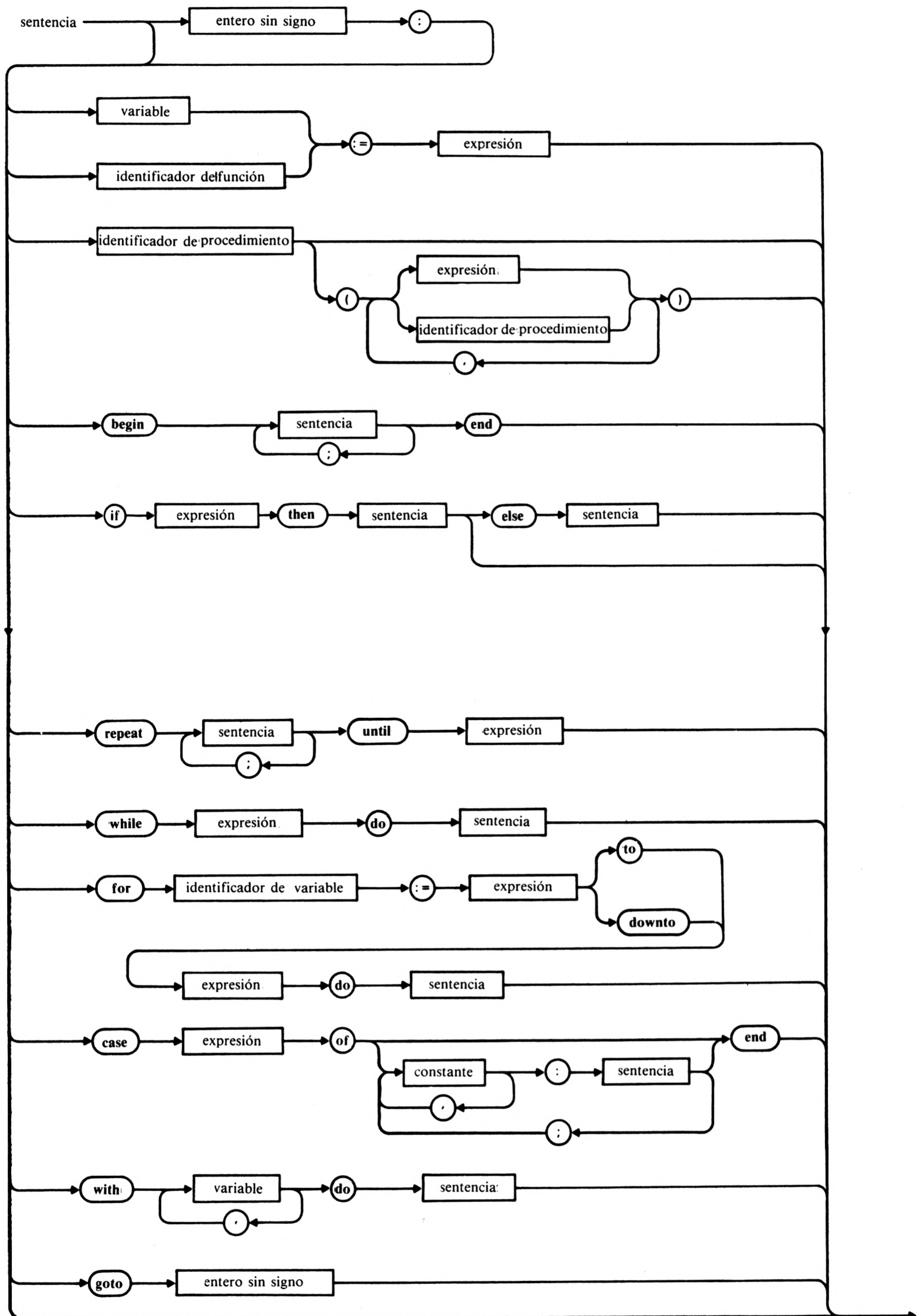

encabezamiento
procedure
function
identificador
lista de parametros
:
identificador de tipo


lista de parámetros
(
var
identificador
,
:
identificador de tipo
encabezamiento
;
)

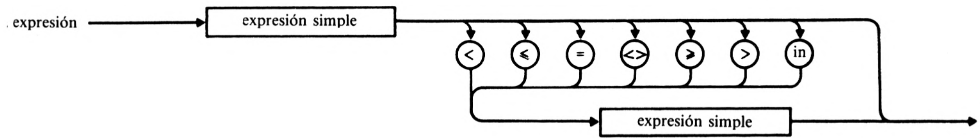

tipo:
tipo simple
↑
identificador de tipo
packed
set
of
array
(
,
)
tipo
record
lista de campos
end
file

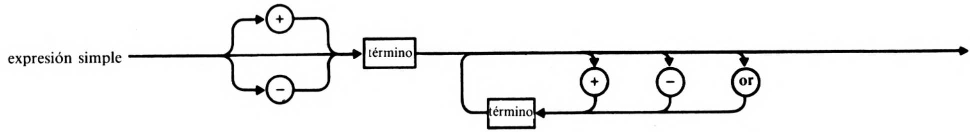

lista de campos
identificador
,
:
tipo
;
case
identificador de tipo
of
constante
(
)

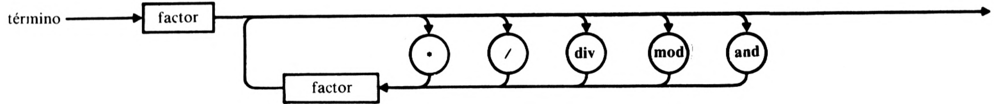

tipo simple
identificador de tipo
(
identificador
,
)
constante
..

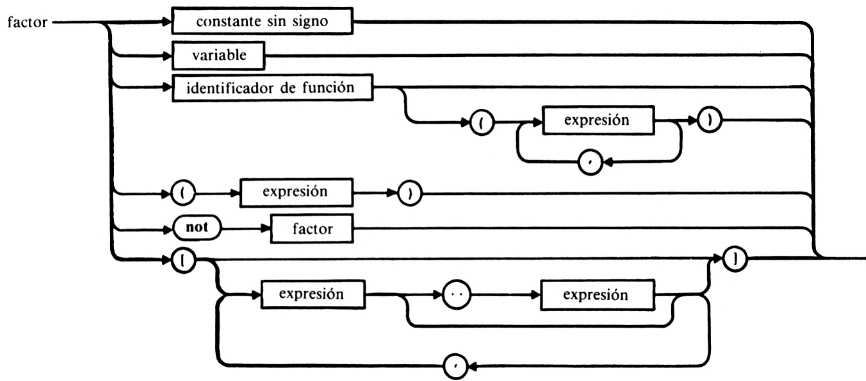


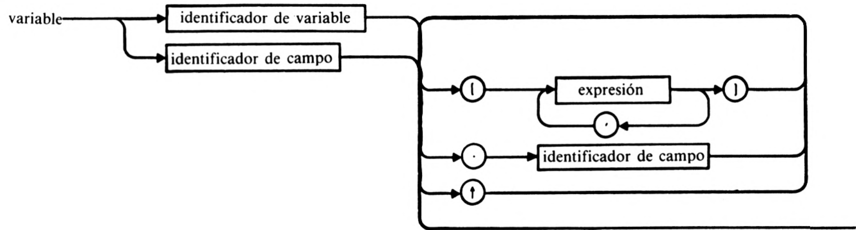
sentencia
entero sin signo
:
variable
identificador de función
:=
expresión
identificador de procedimiento
(
expresión
identificador de procedimiento
)
,
begin
sentencia
;
end
if
expresión
then
sentencia
else
sentencia
repeat
sentencia
;
until
expresión
while
expresión
do
sentencia
for
identificador de variable
:=
expresión
to
downto
expresión
do
sentencia
case
expresión
of
end
constante
,
:
sentencia
;
with
variable
,
do
sentencia
goto
entero sin signo

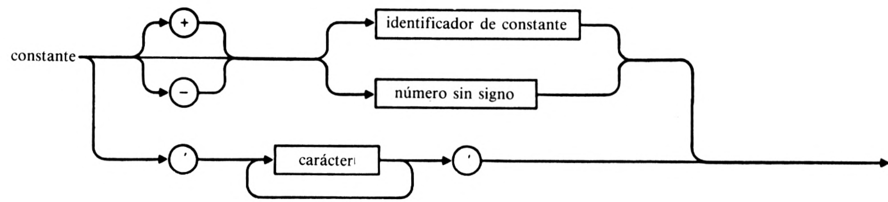

expresión
expresión simple
<
≤
=
<>
≥
>
in
expresión simple

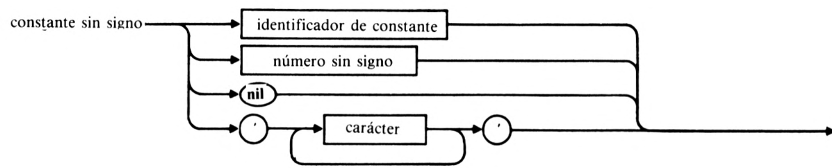

expresión simple
+
−
término
+
−
or
término

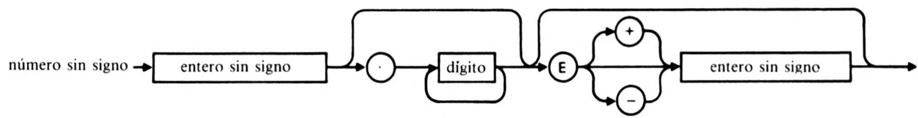

término
factor
·
/
div
mod
and
factor

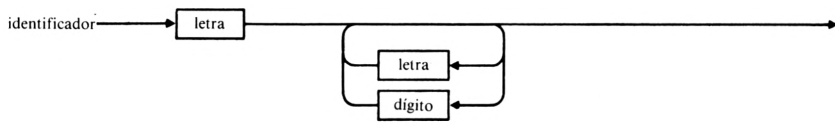

factor
constante sin signo
variable
identificador de función
(
expresión
)
,
(
expresión
)
not
factor
[
expresión
..
expresión
]
,

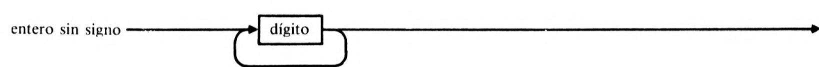

variable
identificador de variable
identificador de campo
[
expresión
]
,
.
identificador de campo
↑

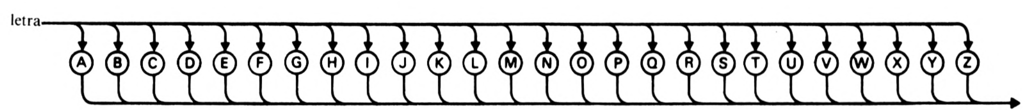

constante
+
−
identificador de constante
número sin signo
'
carácter
'

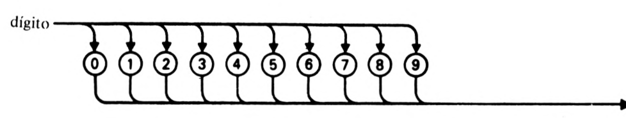

constante sin signo
identificador de constante
número sin signo
nil
'
carácter
'


número sin signo
entero sin signo
.
dígito
E
+
−
entero sin signo

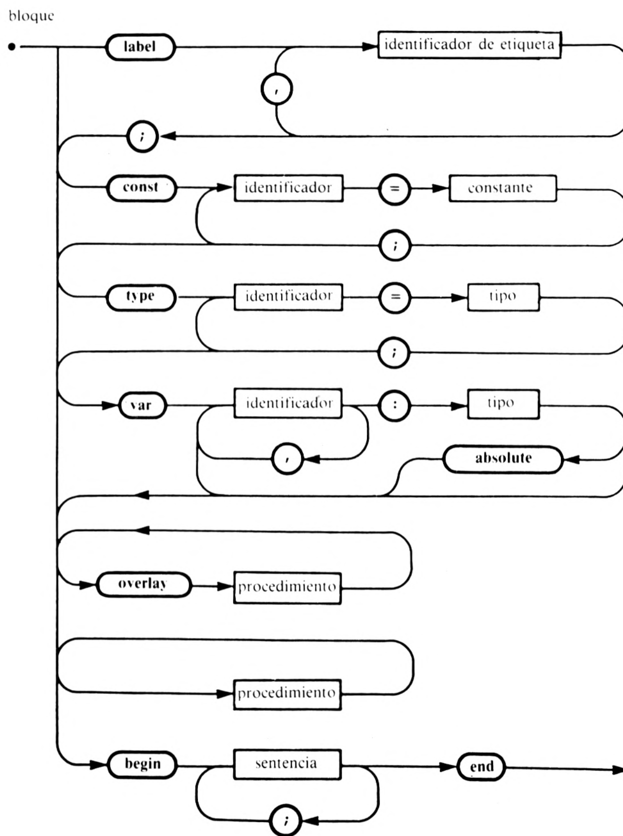

identificador
letra
letra
dígito

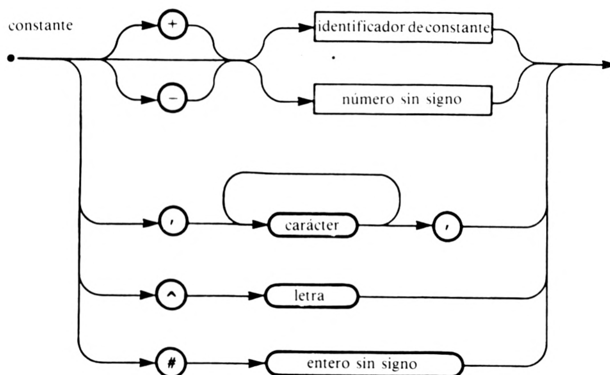

entero sin signo
dígito

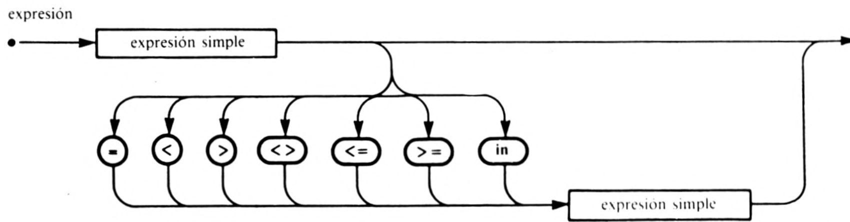

letra
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

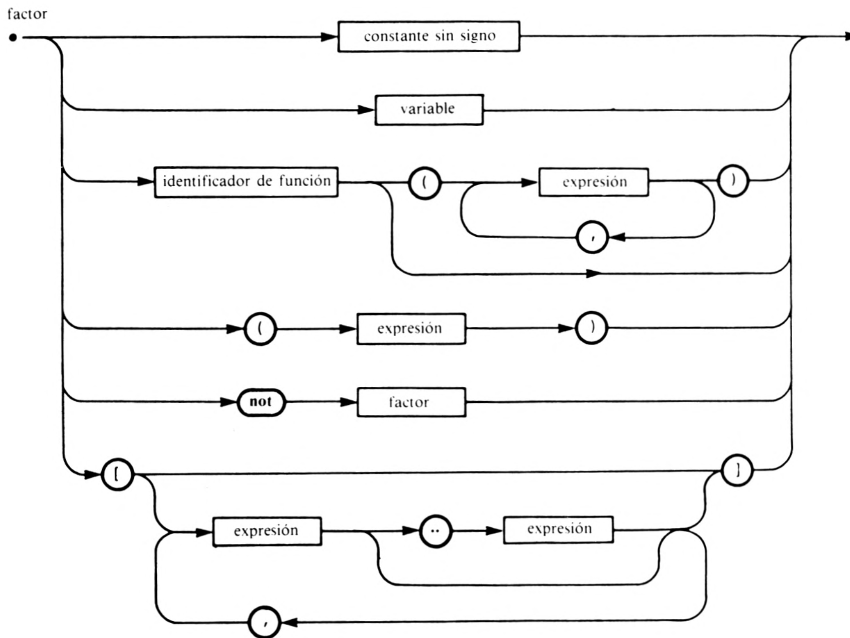

dígito
0 1 2 3 4 5 6 7 8 9

Diagramas de sintaxis del TURBO PASCAL

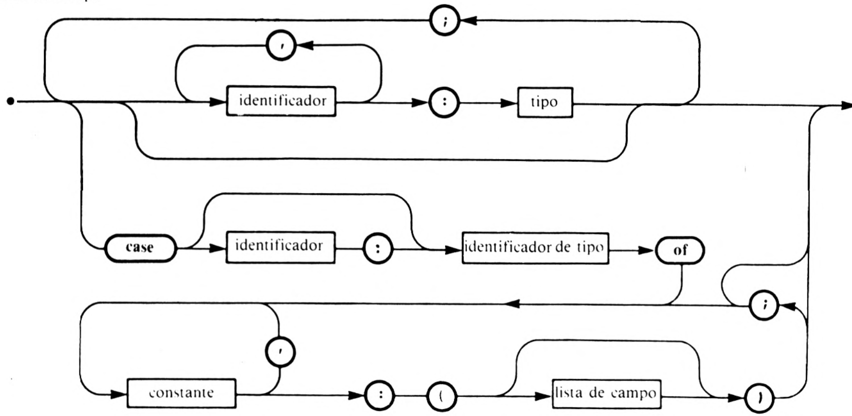




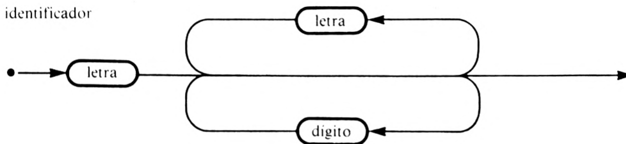

expresión
expresión simple
=
<
>
<>
<=
>=
in
expresión simple


factor
constante sin signo
variable
identificador de función
(
expresión
)
,
(
expresión
)
not
factor
[
]
expresión
..
expresión
,

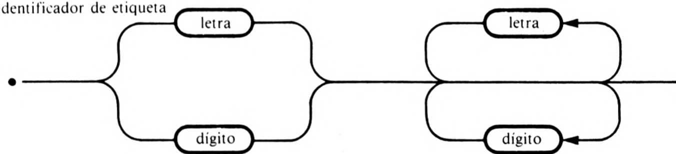
lista de campo



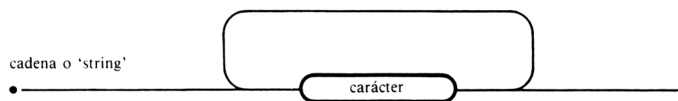
identificador



identificador de etiqueta



cadena o 'string'



lista de parámetros

(; identificador : identificador de tipo)
var ,

programa

program identificador (identificador (; bloque .
,

procedimiento

; bloque ;
procedure identificador lista de parámetros
function identificador lista de parámetros
: identificador de tipo

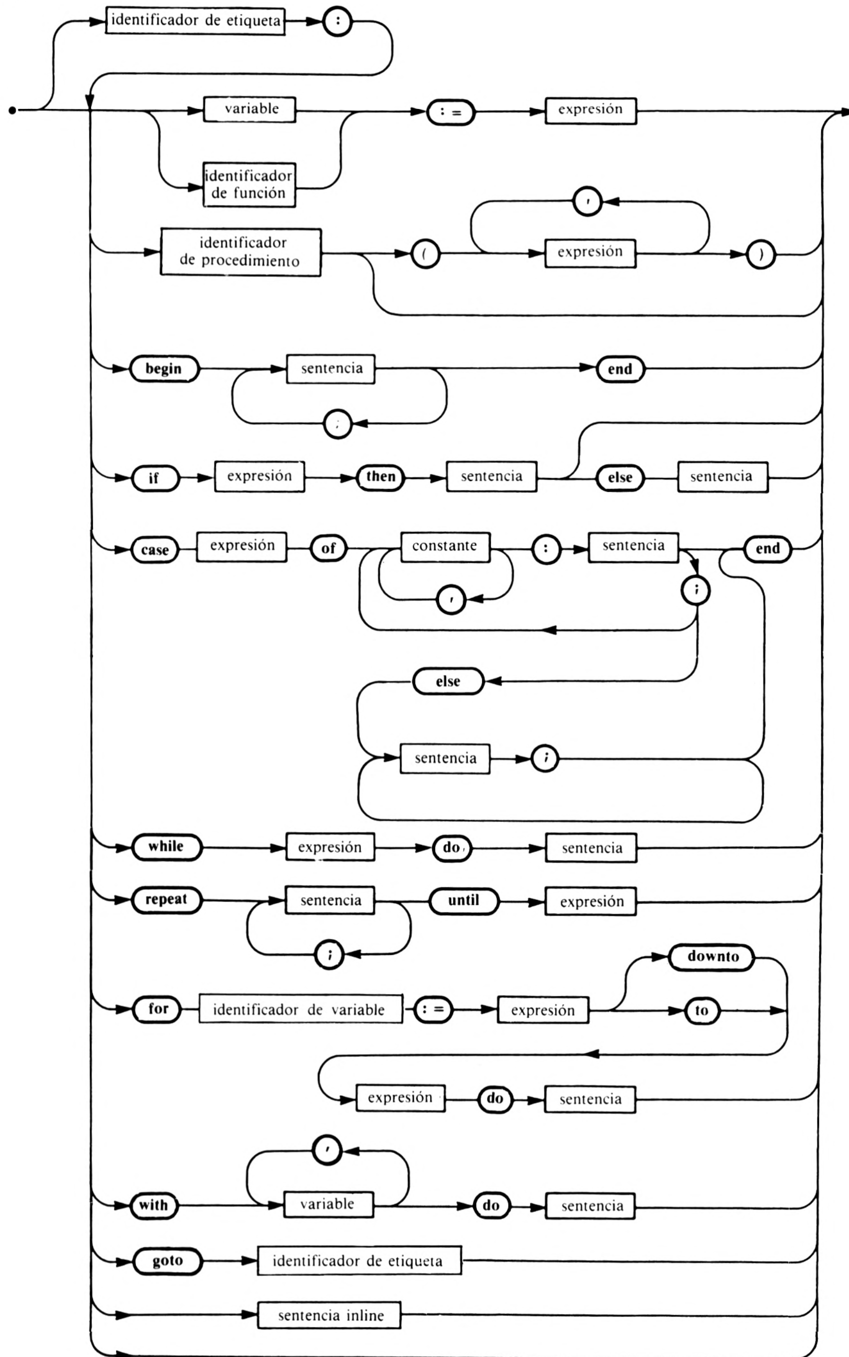
expresión simple

+ − término
+ − or xor
término

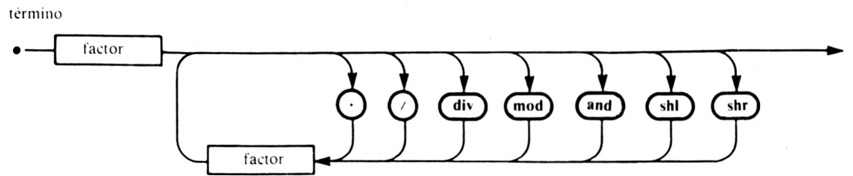
tipo simple

identificador de tipo
(identificador)
:
constante .. constante

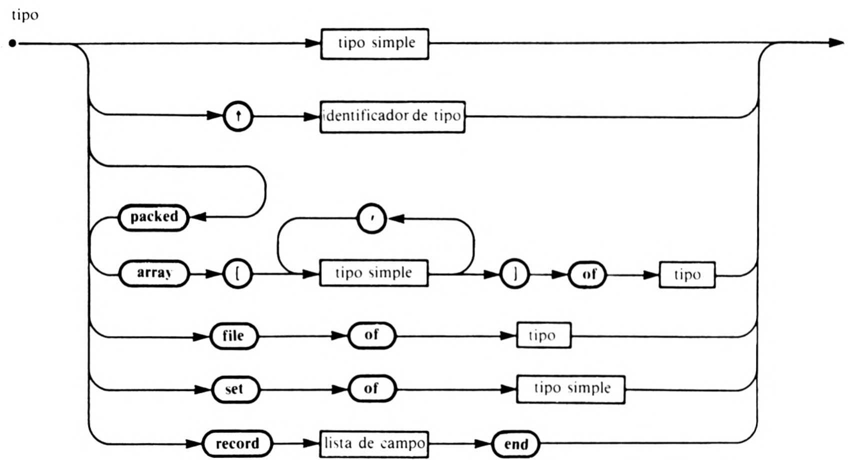
sentencia



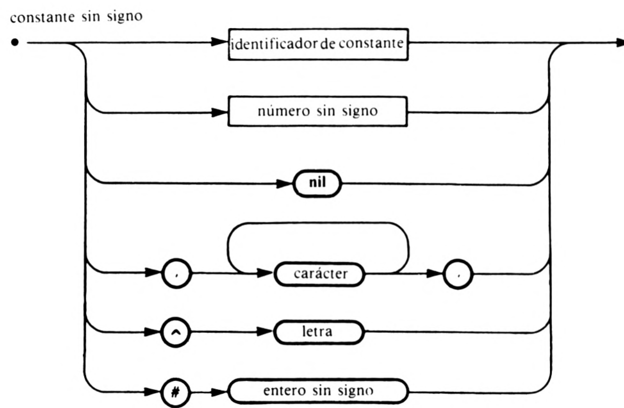
término



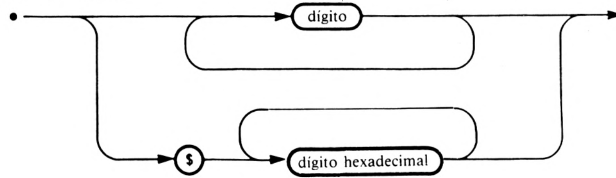
tipo



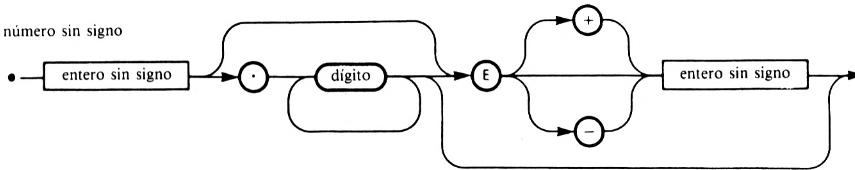
constante sin signo



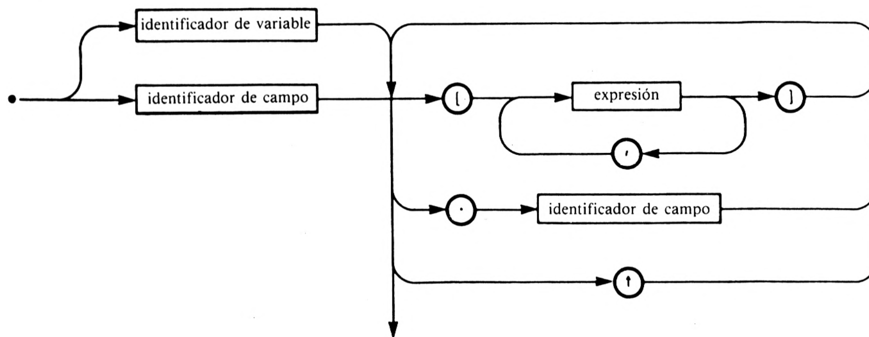
entero sin signo



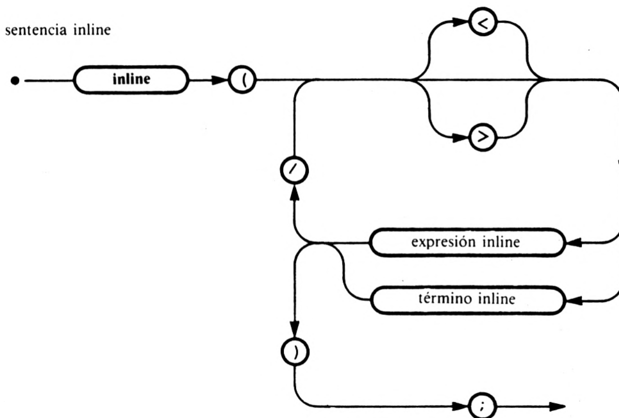
número sin signo



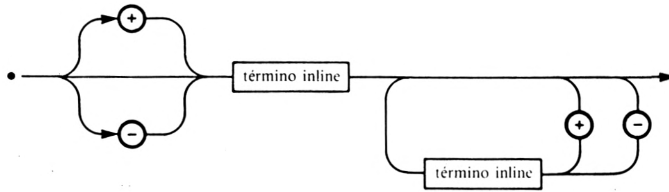
variable

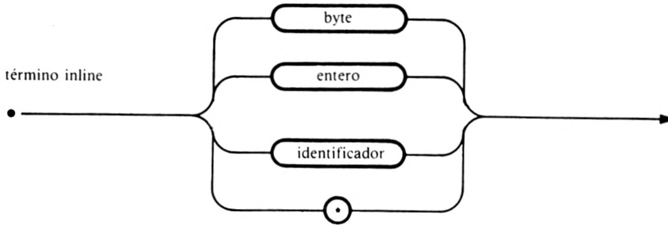


sentencia inline



expresión inline





Indice alfabético

869704, Sr., 38, 66, 116, 156, 178, 187.

abajo arriba (*bottom up*), 105.
ABS, 69.
abs, 69.
abstracción, 146.
acceso aleatorio, 50, 161, 168.
Ackermann, función de, 26.
activa, rutina, 94-5.
ADA, 30.
adición de elementos, 211.
agregaalista, 212.
ALGOL 60, 29-30.
almacenamiento o memoria, 156, 197-217.
ámbito, 92-5.
and, 188.
and, 110.
anidamiento, 121, 143-4, 148, 151.
ANSI, 31-2, 41.
archivo, 48, 49, 50, 86, 114, 136, 155.
  como parámetro, 166-7.
  de texto, 156-7, 163, 168.
  nombre, 50, 159, 160.
  operaciones, 167-8.
  real, 159-61.
  secuencial (véase archivo serie).
  serie, 50, 161, 163.
  sistema de, 48-9.
  temporal, 50, 161.
  variable, 156-7.
argumento, 85-7, 103-4.
array, 96, 119-131, 187, 198.
  como argumento, 104, 131.
  de arrays, 126.
  límites de, 120.
  local, 131.
  tamaño fijo, 130.
arriba abajo (*top down*), 105.
ASC, 115.
ASCII, 112.
asignación, 20, 24, 63, 80, 89, 96, 114, 125, 136, 141, 204-5.
Atkinson, L. V., 59.
atributo de archivo, 50.

backup, 200.
Barron, D. V., 231.

base, tipo, 187, 190.
BASIC estándar, 31.
batch, lenguaje, 46, 47, 59.
BCPL, 30, 90, 152.
begin, 18, 19, 72, 76, 86.
biblioteca, 106, 221-5.
binario, archivo, 156, 157.
Bishop, J. M., 136.
bits, cadena de, 187.
bloques, estructura en, 30, 96, 197-8.
Boole, G., 71, 109.
Boolean, 109-10, 120, 159.
Booleana, 71, 109-11, 122, 187.
bucle, control, 116.
bucle nulo, 122.
buffer, 174.
Buzz, Sra., 37, 39, 78, 91, 96, 141.

C, lenguaje, 30.
cadena o *string*, 112, 132-6, 159.
  constante de, 134.
  entrada de, 169.
campo, 140.
  identificador de, 140-1.
  variantes de, 149.
canal, número de, 156-7.
carácter:
  datos de, 112-5.
  de control, 52.
  de espacio, 169, 172.
caracteres, juegos de, 112, 115, 134.
case:
  etiqueta, 78.
  registros, 150.
  selector, 78.
  sentencia, 77.
CHAIN, 224.
char, 112, 120, 133, 157, 185.
  constante, 112.
  variable, 112.
CHR, 115.
chr, 115.
cláusula, 22.
COBOL, 36, 182.
código fuente, 221-5.
código objeto, 221-5.
colgante, puntero, 203.
columna, 122.
comando u orden, 25, 47, 50.
  de edición, 55.
comentario, 58, 64.
comillas dentro de una cadena, 58.
compatible, tipo, 67.
compilación y ejecución, 57.
compilación, tiempo de, 33.
compilador, 25, 30, 32.
complejo, número, 145.
componentes:
  de archivo, 156.
  tipos de datos de, 120.
compuesta, sentencia, 22.
concatenación, 135.
cònjuntos, 185-194.
  constructor de, 186-7, 190.
  unión de, 188.
const, 79-80, 123.
constante, 39, 69, 78-80, 158.
COPAS, 59.
corchetes, 186.
cos, 69.
count, 76.
cuenta de líneas, 55.
cursor, 55, 182.

DATA, 79.
datos:
  abstracción de, 147.
  estructura de, 141, 206.
  tipo de, 37, 65-8, 79, 86, 109, 113-5, 120, 139, 147, 157-8, 180-1, 185-6, 199, 200-1.
declaración, 19, 79-80, 90-3.
default (véase defecto u omisión).
defecto u omisión, archivo por, 86, 156, 160.
delete, 214.
depuración (*debugging*), 34, 39, 58, 215.
depurador, (*debugger*), 48, 58, 228.
desfasamiento, 175.
diferencia de conjuntos, 188.
DIM, 79, 119.
dinámica, variable, 198.
dinámico, almacenamiento, 198.
directorio, 49.
disciplina, 38, 41.
discriminador, 150, 152-3.
display, 55-6.
dispose, 203, 208, 214.
disposición, 64.
dispositivos, independencia de, 52.
div, 70.
división, 70.
downto, 75.
dump, 58.

EBCDIC, 116.
ecount, 120-1.
editor, 25, 45-50, 53-6.
ejecución:
  método de, 32.
  velocidad de, 33.
else, 25, 71.

encabezamiento de programa, 86-7, 159, 160.
END, 78.
end, 21-3, 72, 76, 86.
end-of-file o final de archivo, 171-2, 176.
enlazador, 222.
ensamblador, lenguaje, 187, 201.
entrada de datos, 52.
eof, 164-5, 169, 172.
eoln, 169.
equivalencia de nombre, 125, 153.
error de sintaxis, 192.
errores, 38-9, 57.
  detección de, 122.
escalar, tipo, 66.
escribedía, 181.
espaciado, 64.
espacio, carácter de, 158.
espacio de memoria, 94.
estructura, 21, 37.
  (véase también Sra. Buzz).
etiqueta, 74, 78.
Euclides, 99.
evaluación, 40.
evaluación, función, 147-8.
EXP, 69.
exp, 69, 70.
expresión aritmética, 70.
expresión de relación, 71.
extensión o ampliación, 231.
external files, 159-61.
EXECUTE, 58, 160.

false o falso, 71, 110, 122, 159.
FILE, 159.
FOR, 22, 74.
for, bucle, 97.
for, sentencia, 21-2, 74-5.
formato de salida, 177-80.
fuente, código, 221-5.
fuente, lenguaje, 59.
función, 83.

gash value, 40.
get, 162-4.
global, 197-8, 207.
GOSUB, 83.
GOTO, 25, 73-4.
goto, sentencia, 73-4.
gráficos, 64, 77, 182.

hardware, 35.
heap, 200.
HELP, 53.
herramientas, 105-6.
Hoare, C. A. R., 153.
housekeeping, 211, 217.
howout, 68, 78.

identificador, 64-5, 68.
  de campo, 141.
IF, 71-3.
IF END, 165.
if, sentencia, 71-3, 75.
implementación del Pascal, 30, 193.
indicador (véase puntero electrónico).
input, 157-8, 164.
in, 189.
include, 225.
incorporada, función, 68, 83.
incorporado, procedimiento, 90.
indexado, tipo de datos, 120.
índice de *array* o matriz (véase subíndice).
información, ocultación de, 91, 147.
informe Pascal, 31, 104, 147, 149, 221.
inicialización (de archivos), 164.
INPUT, 23, 79, 156.
input/output, 86, 157-64, 173-7.
INT, 69, 89.
integer, 20-1, 65, 68, 79, 120, 157.
interactivo, 25, 46.
  entrada/salida, 173-7.
  lenguaje, 59.
intérprete, 32.
intersección de conjuntos, 188.
invitación (*prompt*), 174.

Jensen, K., 31.

Kaye, D. R., 175.
Kemeny, J. G., 83.
Kernighan, B. W., 105.
Knuth, D. E., 206.
Kurtz, T. E., 83.

lap, 123.
Ledgard, H. F., 42.
LEFT, 135.
legibilidad, 34-6, 65, 101-2, 151, 227.
lenguaje estructurado en bloques, 30, 96.
LET, 24.
liberación de espacio de memoria o almacenamiento, 202.
lista, 209, 218.
listado, 58.
llaves, 64.
local:
  ámbito, 92.
  declaración, 90, 92-3.
  manual, 104, 135, 163, 182.
  rutina, 91.

tiempo de vida, 94-5.
variable, 94-97, 100-1.
LOG, 69.
longitud de cadena, 133-5.
longitud de parte fraccionaria, 178-9.

mando:
estado de, 52.
lenguaje de, 48, 50.
mantenimiento, 34-5, 39, 97.
MAT, 125.
matriz (véase *array*).
máximo común divisor, 83.
maxint, 67, 69.
mayúsculas, letras, 17-8.
memoria, 94.
tamaño de la, 33.
microordenador, 45.
minúsculas, letras, 17-8.
mod, 70.
montón (véase pila).
Mudd, W., 26, 38, 60, 34, 135, 152.
multilínea, función, 83.
múltiple, asignación, 63.

Nevison, J. M., 42.
new, 202-3.
nil, 202, 207, 215.
nombre, 104.
nombre de tipo, 94, 126.
North, S. D., 59.
not, 110.
numérico, 20.
número de línea, 25, 54.

objetivos del Pascal, 29.
ON, 77.
operador, 20, 110, 114, 187.
OPTIONBASE, 79.
or, 188.
or, 110.
ord, 115.
ordenación:
de declaraciones, 79, 84, 102.
de valores, 114.
output, 157-8, 164.

packed, 131.
parámetro, 85-6, 166.
formal, 85.
matricial, 130.
real, 85.
de valor, 104.
de variable, 67, 104.
Parnas, D. L., 91.
Pascal, norma, 31, 104, 130.
PEE, 46.
Perkins, 40, 87, 122-4, 148, 151-2, 192, 201-2, 222.
piggy, 102.
pila, 124, 147, 199.
PL/I, 30.
Plauger, P. J., 105.
POKE, 77.
polimórfico, 20, 113.
portabilidad, 32, 135, 168.
potencia, 88.
precedencia, 114.
pred, 115.
preparación, 47-8, 230.
préstamo de memoria, 202.
prettyprinting (escritura bonita), 23.
Primple, M. d'A., 26, 37, 42, 72-3, 80, 132, 134, 210.
procedimiento, 83, 89-90, 162, 170, 181.
encabezamiento, 89.
programa, 21, 159.
programa de clasificación (*sorting*), 161.
programa objeto, 48, 58.
programa principal, 94.
programación, estilo de, 104-5, 208.
programación estructurada, 37.
progresivo, refinamiento, 91.
prompts, 174.
pudding de carne y riñones, 217.
puntero, 200-1.
puntero adicional, 217.
puntero electrónico, 55.
puntero errante, 209.
punto de exploración, 54.
punto y coma, 24, 72.
put, 162-4.

RANDOMIZE, 79.
raw mode (modo no elaborado), 174.
read, 23, 157, 158, 162, 172.
READ, 24, 79.
readcubed (leecubo), 88, 97.
readln, 170-1.
real, 20, 65, 69, 79, 157.
rebanada o *slice* de *array*, 126.
refinamiento progresivo, 91.
regristro (*record*), 139-49, 182, 198, 208.
registro con variantes, 149-53.
reglas de precedencia, 114.
relación:
expresión de, 71.
operador de, 71, 113-4, 189.
repeat, sentencia, 76, 88.
repetición, 98, 201.
repetición mutua (recurrencia), 98.
Reps, T., 59.

reservada, palabra, 65.
reset, 161-164.
RESTORE, 79.
restricción en Pascal, 193.
resultado de la función, 86.
RETURN, 83, 89.
rewrite, 161, 164.
Richards, M., 152.
RIGHT, 135.
RND, 69, 79.
Round, F., 42, 99, 111.
roving pointers, 209.
RUN, 58, 160.
run-time, 33.
run-time error, 58.
ruptura, 58.
  autónoma, 91.
  incorporada, 90.
rutina, 90.

Sale, A., 136.
SAVE, 26, 58, 161.
SEG, 135.
seguimiento de la ejecución (*tracing*), 59.
seguridad (véase también Perkins), 40.
semi-interactivo, 25, 59.
sentencia, 71-9.
  de bucle, 73.
sentencia nula, 24.
set of, 186, 193.
SGN, 69.
SIN, 69.
sin, 69.
sistema:
  de archivo, 58.
  de ejecución (*run time system*), 32.
  operativo, 25, 45, 48, 160.
  de representación visual, 56-7.
slices, 126.
Sneeringer, W. J., 153.
software, 35, 225.
sonidos, 77.
sorting, 161.
SQR, 69.
sqr, 69, 70.
sqrt, 69.
stack (véase pila).
Standish, T. A., 206.
STOP, 78.
string o cadena, 20, 37, 58, 132-6.
strongly typed, 202.
subcadena (substring), 20, 55.
subíndice, 40, 120.
subprograma, 37.
subrango, 66-7, 104, 113, 115, 122, 185-6.
subrutina, 83.
succ, 115.

TAN, 69.
Teitelbaum, T., 59.
Teitelman, W., 56.
terminal, 147, 157, 160.
text, 156-7.
texteof, 166, 173.
textfile, 156.
THEN, 71.
then, 71.
tiempo compartido, 49.
tiempo de ejecución, 33.
tipo de datos sencillo, 109, 113, 153.
tipo definido por el usuario, 113-4, 185-6.
tipo escalar (véase tipo definido por el usuario).
tipo estructurado (véase registro).
tracing, 59.
traducción de conceptos, 63.
true, 71, 110, 122, 159.
trunc, 69-70.
type, 66, 68, 80, 122, 153.

unión de conjuntos, 188.
unión indiscriminada, 152.
unpack, 132.
USING, 177.
útiles (véase herramientas).

vacío, conjunto, 186.
var, 66, 68, 80, 131, 156, 161.
variable, 20.
  no asignada, 122.
  dinámica, 206-8.
  ordinaria, 200.
variaciones en Basic y en Pascal, 31.
variantes, 151.
ventajas del Pascal, 29-30, 227-8.
ventana en un archivo, 162-3, 172, 205.
verificación, 39, 193.

Welsh, 153.
while, 75-6.
Whitby-Strevens, C., 152.
Wirth, N., 30-1, 91, 142.
with, sentencia, 147-8.
write, 157, 162.
writeln, 24, 90, 170-1.
writeonly, 159.

zorro, 102.

El PASCAL es uno de los lenguajes más aptos para el desarrollo de aplicaciones profesionales y para el trabajo tanto con pequeños microordenadores como con grandes sistemas.

PASCAL A PARTIR DEL BASIC está escrito para el programador razonablemente competente en BASIC, que no necesita que le expliquen conceptos elementales, sino que le ayuden a adaptar su forma de pensar desde el BASIC al PASCAL.

Este libro, por tanto:

- Enseña a pensar y programar en PASCAL, aprovechando los conocimientos actuales de programación.
- Desarrolla las ventajas de trabajar en un lenguaje estructurado.
- Aclara los conceptos (sistema operativo, editor, etc.) en el entorno PASCAL.
- Muestra cómo se resuelven en PASCAL problemas difíciles o engorrosos de analizar en BASIC.
- Contiene, además, los diagramas de sintaxis del PASCAL estándar y de una de las versiones más difundidas: el TURBO PASCAL.

PASCAL A PARTIR DEL BASIC posee un eficaz planteamiento didáctico y está escrito en un tono ligero, que hace divertida su lectura, lo que le convierte en un libro idóneo para iniciarse en la programación en PASCAL.

ANAYA
MULTIMEDIA

PASCAL a partir del BASIC
Peter Brown

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/

[FRA] Ce document a été préservé numériquement à des fins éducatives et d'études, et non commerciales.
[ENG] This document has been digitally preserved for educational and study purposes, not for commercial purposes.
[ESP] Este documento se ha conservado digitalmente con fines educativos y de estudio, no con fines comerciales.