

# **PROGRAMACIÓN CON PASCAL**

## **JOHN KONVALINA**

Professor of Mathematics  
and Computer Science  
University of Nebraska at Omaha

## **Stanley Wileman**

Associate Professor  
of Mathematics and Computer Science  
University of Nebraska at Omaha

### **Traductor**

Roberto Luis Escalona García  
Maestro en Ciencias, UNAM

UNIVERSIDAD DE LA REPÚBLICA  
FACULTAD DE INGENIERÍA  
DPTO. DE DOCUMENTACION Y BIBLIOTECA  
BIBLIOTECA CENTRAL  
Ing. Edo. García de Zúñiga  
MONTEVIDEO - URUGUAY

No. de Entrada 0518

3.4-97

### **Revisión Técnica:**

Jorge Agustín Olvera Rodríguez  
Doctor en la Universidad de Austin, Texas  
Director del Depto. de Ciencias Computacionales,  
Instituto Tecnológico de Estudios Superiores Monterrey,  
(Campus Monterrey).

**SIBUR**

**McGRAW-HILL**

MÉXICO • BOGOTÁ • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA  
MADRID • NUEVA YORK • PANAMÁ • SAN JUAN • SANTIAGO • SÃO PAULO  
AUCKLAND • HAMBURGO • LONDRES • MILÁN • MONTREAL • NUEVA DELHI  
PARÍS • SAN FRANCISCO • SINGAPUR • ST. LOUIS • SIDNEY • TOKIO • TORONTO

## **PROGRAMACION CON PASCAL**

Prohibida la reproducción total o parcial de esta obra,  
por cualquier medio, sin autorización escrita del editor.

**DERECHOS RESERVADOS © 1989, respecto a la primera edición en español por  
McGRAW-HILL/INTERAMERICANA DE MEXICO, S. A. DE C. V.**

Atlacomulco 499-501, Fracc. Ind. San Andrés Atoto

53500 Naucalpan de Juárez, Edo. de México

Miembro de la Cámara Nacional de la Industria Editorial, Reg. Núm. 1890

**ISBN 968-422-295-5**

Traducido de la primera edición en inglés de  
**PROGRAMMING WITH PASCAL**

Copyright © MCMLXXXVII, by McGraw-Hill, Inc., U. S. A.

**ISBN 0-07-035224-0**

1234567890              P.E.-89              8012345679

Impreso en México

Printed in Mexico

Esta obra se terminó de  
imprimir en diciembre de 1988  
en Programas Educativos, S. A.  
Calz. Chabacano No. 65-A  
Col. Asturias  
Delegación Cuauhtémoc  
06860, México, D. F.

Se tiraron 4 000 ejemplares

# ACERCA DE LOS AUTORES

*John Konvalina* es profesor de matemáticas y ciencias de la computación en la Universidad de Nebraska en Omaha. Obtuvo el grado de doctor en la Universidad del Estado de Nueva York en Buffalo. Ha escrito dos libros de texto y varios artículos acerca de combinatórica y enseñanza de ciencias de la computación.

*Stanley Wileman* es profesor asociado de matemáticas y ciencias de la computación en la Universidad de Nebraska en Omaha. Obtuvo el grado de maestro en ciencias de la computación en la Universidad de Houston con Elliott Organick, ya fallecido. El profesor Wileman ha impartido cursos sobre ciencias de la computación durante más de una década y ha colaborado en seis artículos sobre enseñanza de ciencias de la computación. Es miembro activo de la Association for Computer Machinery.

A la memoria de nuestros padres

# CONTENIDO EN BREVE

<i>Prefacio</i>	xxi
1 Introducción a la computación	1
2 Introducción al PASCAL	29
3 Entrada, salida y resolución de problemas en Pascal	81
4 Diseño descendente y procedimientos elementales	121
5 Selección	171
6 Ciclos	213
7 Procedimientos y funciones	265
8 Tipos de datos	319
9 Arreglos	359
10 Procesamiento de cadenas de caracteres	411
11 Registros	461
12 Archivos	499
13 Introducción a las estructuras de datos	533
Apéndices	575
Glosario	591
Respuestas a los ejercicios	605
Índice	637



# CONTENIDO

<b>Prefacio</b>	xxi
<b>CAPÍTULO 1 INTRODUCCIÓN A LA COMPUTACIÓN</b>	1
<i>Objetivos</i>	2
<i>Panorama general del capítulo</i>	2
SECCIÓN 1.1 ORGANIZACIÓN DE LAS COMPUTADORAS	2
Entrada/salida	3
Unidad de memoria	4
Unidad central de procesamiento	4
SECCIÓN 1.2 LENGUAJES DE COMPUTACIÓN	5
Instrucciones de computadora	5
Lenguaje de máquina	6
Lenguajes de bajo nivel	7
Lenguajes de alto nivel	8
Pascal	8
Compilación y ejecución	9
SECCIÓN 1.3 RESOLUCIÓN DE PROBLEMAS	10
Análisis del problema	11
Diseño de algoritmos	12
Algoritmos	14
Ejecución de algoritmos	15

Resolución por computadora	20
Programa en Pascal	20
Compilación y ejecución de programas	21
Prueba de programas	22
<b>SECCIÓN 1.4 TÉCNICAS DE PRUEBA Y DEPURACIÓN</b>	<b>22</b>
<b>SECCIÓN 1.5 REPASO DEL CAPÍTULO</b>	<b>23</b>
 <i>Avance del capítulo 2</i>	 24
<i>Palabras clave del capítulo 1</i>	25
<i>Introducción a los ejercicios y problemas para resolución en computadora</i>	25
<i>Ejercicios del capítulo 1</i>	26
<i>Especificación de problemas para resolución en computadora</i>	27
 <b>✓ CAPÍTULO 2 INTRODUCCIÓN AL PASCAL</b>	 29
<i>Objetivos</i>	30
<i>Panorama general del capítulo</i>	30
<b>SECCIÓN 2.1 EJEMPLO EN PASCAL</b>	<b>30</b>
Estructura de un programa en Pascal	34
<b>SECCIÓN 2.2 IDENTIFICADORES, CONSTANTES Y VARIABLES</b>	<b>35</b>
Identificadores en Pascal	35
Diagramas de sintaxis	36
Constantes	37
Variables	39
Otro programa en Pascal	43
Ejercicios de la sección 2.2	45
<b>SECCIÓN 2.3 TIPOS DE DATOS SIMPLES: ENTEROS, REALES, BOOLEANOS Y DE CARACTERS</b>	<b>46</b>
Tipo de datos enteros (integer)	48
Declaración de variables enteras	48
Tipo de datos reales (real)	49
Declaración de variables reales	50
Tipo de datos de caracteres (char)	51
Declaración de constantes y variables de caracteres	52
Tipo de datos booleanos (Boolean)	52
Ejercicios de la sección 2.3	53
<b>SECCIÓN 2.4 PROPOSICIONES DE ASIGNACIÓN Y EXPRESIONES ARITMÉTICAS</b>	<b>54</b>
Asignación	54
Expresiones aritméticas	56
Proposiciones de asignación y expresiones aritméticas	59
Orden de las operaciones aritméticas	60
Traducción de expresiones algebraicas a Pascal	62
Funciones estándar (incluidas)	63
Ejercicios de la sección 2.4	67

<b>SECCIÓN 2.5</b>	<b>TÉCNICAS DE PRUEBA Y DEPURACIÓN</b>	<b>69</b>	<b>xi</b>
Introducción, modificación y ejecución de programas en Pascal		70	CONTENIDO
<i>Recordatorios de Pascal</i>		70	
<b>SECCIÓN 2.6</b>	<b>REPASO DEL CAPÍTULO</b>	<b>72</b>	
<i>Referencias de Pascal</i>		72	
<i>Avance del capítulo 3</i>		74	
<i>Palabras claves del capítulo 2</i>		75	
<i>Ejercicios del capítulo 2</i>		75	
<i>Problemas del capítulo 2 para resolución en computadora</i>		77	
 <b>CAPÍTULO 3 ENTRADA, SALIDA Y RESOLUCIÓN DE PROBLEMAS EN PASCAL</b>		 <b>81</b>	
<i>Objetivos</i>		82	
<i>Panorama general del capítulo</i>		82	
<b>SECCIÓN 3.1</b>	<b>ENTRADA Y SALIDA</b>	<b>82</b>	
Entradas numéricas		83	
Entrada de caracteres		86	
Entrada mixta: caracteres y números		87	
Salida		88	
Salida con formato		91	
Anchuras de campo explícitas para expresiones enteras		92	
Anchuras de campo explícitas para expresiones de caracteres y cadenas		93	
Anchuras de campo explícitas para expresiones reales		93	
Ejercicios de la sección 3.1		95	
<b>SECCIÓN 3.2</b>	<b>RESOLUCIÓN DE PROBLEMAS CON PASCAL</b>	<b>99</b>	
<b>SECCIÓN 3.3</b>	<b>INTRODUCCIÓN A LA ENTRADA Y SALIDA DE ARCHIVOS DE TEXTO (OPCIONAL)</b>	<b>106</b>	
<b>SECCIÓN 3.4</b>	<b>TÉCNICAS DE PRUEBA Y DEPURACIÓN</b>	<b>108</b>	
Errores de entrada		108	
Errores de salida		110	
<i>Recordatorios de Pascal</i>		111	
<b>SECCIÓN 3.5</b>	<b>REPASO DEL CAPÍTULO</b>	<b>111</b>	
<i>Referencias de Pascal</i>		112	
<i>Avance del capítulo 4</i>		112	
<i>Palabras clave del capítulo 3</i>		113	
<i>Ejercicios del capítulo 3</i>		113	
<i>Problemas del capítulo 3 para resolución en computadora</i>		114	

<i>Objetivos</i>	122
<i>Panorama general del capítulo</i>	122
SECCIÓN 4.1 INTRODUCCIÓN A LOS PROCEDIMIENTOS Y AL DISEÑO DESCENDENTE	122
Estructura de los procedimientos	127
Invocación y ejecución de los procedimientos	128
Cuándo conviene emplear procedimientos	132
Colocación de los procedimientos y tipos rígidos	134
SECCIÓN 4.2 RESOLUCIÓN DE PROBLEMAS MEDIANTE PROCEDIMIENTOS SIMPLES	134
Ejercicios de la sección 4.2	138
SECCIÓN 4.3 PROCEDIMIENTOS CON PARÁMETROS	140
Invocación de procedimientos con parámetros	143
Variables locales y globales	147
Ejercicios de la sección 4.3	148
SECCIÓN 4.4 RESOLUCIÓN DE PROBLEMAS MEDIANTE PROCEDIMIENTOS CON PARÁMETROS	150
Análisis de problemas	150
Diseño descendente	151
Algoritmo	152
Resolución en la computadora	152
Ejercicios de la sección 4.4	156
SECCIÓN 4.5 DISEÑO Y PRUEBA DESCENDENTES	157
Revisión de ortografía	157
Prueba descendente	160
SECCIÓN 4.6 TÉCNICAS DE PRUEBA Y DEPURACIÓN	162
<i>Recordatorios de Pascal</i>	163
SECCIÓN 4.7 REPASO DEL CAPÍTULO	164
<i>Referencias de Pascal</i>	164
<i>Palabras clave del capítulo 4</i>	165
<i>Avance del capítulo 5</i>	166
<i>Ejercicios del capítulo 4</i>	166
<i>Problemas del capítulo 4 para resolución en computadora</i>	167
<b>CAPÍTULO 5 SELECCIÓN</b>	171
<i>Objetivos</i>	172
<i>Panorama general del capítulo</i>	172

<b>SECCIÓN 5.1</b>	<b>EXPRESIONES BOOLEANAS</b>	174
Variables booleanas		174
Operadores relacionales		175
Operadores booleanos		176
Orden de los operadores		178
Ejercicios de la sección 5.1		180
<b>SECCIÓN 5.2</b>	<b>SELECCIÓN MEDIANTE LA PROPOSICIÓN IF</b>	181
Proposición IF-THEN		183
Proposiciones IF anidadas		186
Ejercicios de la sección 5.2		188
<b>SECCIÓN 5.3</b>	<b>SELECCIÓN MEDIANTE LA PROPOSICIÓN CASE</b>	190
Ejercicios de la sección 5.3		195
<b>SECCIÓN 5.4</b>	<b>RESOLUCIÓN DE PROBLEMAS CON SELECCIÓN</b>	196
<b>SECCIÓN 5.5</b>	<b>TÉCNICAS DE PRUEBA Y DEPURACIÓN</b>	199
Expresiones booleanas		199
Paréntesis		200
Validación		200
<i>Recordatorios de Pascal</i>		201
<b>SECCIÓN 5.6</b>	<b>REPASO DEL CAPÍTULO</b>	202
<i>Referencias de Pascal</i>		203
<i>Avance del capítulo 6</i>		205
<i>Palabras clave del capítulo 5</i>		205
<i>Ejercicios del capítulo 5</i>		205
<i>Problemas del capítulo 5 para resolución en computadora</i>		207
<b>\ CAPÍTULO 6 CICLOS</b>		213
<i>Objetivos</i>		214
<i>Panorama general del capítulo</i>		214
<b>SECCIÓN 6.1</b>	<b>LA PROPOSICIÓN WHILE</b>	216
Centinelas y ciclos		219
Control de ciclos por medio de variables booleanas		221
Proposiciones WHILE anidadas		223
Ejercicios de la sección 6.1		224
<b>SECCIÓN 6.2</b>	<b>OTRAS ESTRUCTURAS CÍCLICAS</b>	226
Proposición REPEAT-UNTIL		227
La proposición FOR		229
¿Estructuras de control definidas o indefinidas?		235
Ejercicios de la sección 6.2		237
<b>SECCIÓN 6.3</b>	<b>RESOLUCIÓN DE PROBLEMAS MEDIANTE CICLOS</b>	239

SECCIÓN 6.4	EXACTITUD DE LOS PROGRAMAS E INVARIANTES DE CICLO (OPCIONAL)	242
	Precondiciones y postcondiciones	243
	Invariantes de ciclo	245
SECCIÓN 6.5	ESTRUCTURAS DE CONTROL FUNDAMENTALES: RESUMEN	247
SECCIÓN 6.6	TÉCNICAS DE PRUEBA Y DEPURACIÓN	250
	Ciclos infinitos	250
	Errores en ciclos	251
	<i>Recordatorios de Pascal</i>	253
SECCIÓN 6.7	REPASO DEL CAPÍTULO	253
	<i>Referencias de Pascal</i>	253
	<i>Avance del capítulo 7</i>	255
	<i>Palabras clave del capítulo 6</i>	255
	<i>Ejercicios del capítulo 6</i>	255
	<i>Problemas del capítulo 6 para resolución en computadora</i>	257
<b>CAPÍTULO 7 PROCEDIMIENTO Y FUNCIONES</b>		265
	<i>Objetivos</i>	266
	<i>Panorama general del capítulo</i>	266
SECCIÓN 7.1	REPASO DE PROCEDIMIENTOS Y PARÁMETROS	267
	Parámetros de valor y variables	268
	Procedimientos anidados	271
	Reglas de alcance	273
	Ejercicios de la sección 7.1	276
SECCIÓN 7.2	FUNCIONES	280
	Funciones estándar	280
	Función <i>Odd</i>	280
	Funciones <i>Eoln</i> y <i>Eof</i>	281
	Funciones definidas por el usuario	284
	Invocación de funciones	286
	Ejercicios de la sección 7.2	289
SECCIÓN 7.3	PROCEDIMIENTOS Y FUNCIONES RECURSIVOS	292
	Recursión e iteración	296
	Ejercicios de la sección 7.3	-
SECCIÓN 7.4	RESOLUCIÓN DE PROBLEMAS MEDIANTE PROCEDIMIENTOS Y FUNCIONES	298
	Problema de juego de multiplicación	298
	Generadores de números aleatorios	300
	Solución completa al problema de juego de multiplicación	302

<b>SECCIÓN 7.5</b>	<b>TÉCNICAS DE PRUEBA Y DEPURACIÓN</b>	<b>305</b>	<b>xv</b>
	Efectos secundarios	306	CONTENIDO
	Declaración forward	306	
	Transportabilidad	308	
<i>Recordatorios de Pascal</i>		308	
<b>SECCIÓN 7.6</b>	<b>REPASO DEL CAPÍTULO</b>	<b>309</b>	
<i>Referencias de Pascal</i>		310	
<i>Avance del capítulo 8</i>		312	
<i>Palabras clave del capítulo 7</i>		312	
<i>Ejercicios del capítulo 7</i>		313	
<i>Problemas del capítulo 7 para resolución en computadora</i>		315	
<b>CAPÍTULO 8 TIPOS DE DATOS</b>		<b>319</b>	
<i>Objetivos</i>		320	
<i>Panorama general del capítulo</i>		320	
<b>SECCIÓN 8.1</b>	<b>TIPOS DE DATOS DEFINIDOS POR EL USUARIO O ENUMERADOS</b>	<b>321</b>	
	Tipos de datos enumerados	321	
	Definiciones TYPE	324	
	Ejercicios de la sección 8.1	326	
<b>SECCIÓN 8.2</b>	<b>TIPOS DE DATOS DE SUBESCALA</b>	<b>328</b>	
	Ejercicios de la sección 8.2	331	
<b>SECCIÓN 8.3</b>	<b>FUNCIONES ORDINALES ESTÁNDAR: PRED, SUCC, CHR Y ORD</b>	<b>333</b>	
	<i>Pred y Succ</i>	333	
	<i>Ord y Chr</i>	334	
	Ejercicios de la sección 8.3	338	
<b>SECCIÓN 8.4</b>	<b>RESOLUCIÓN DE PROBLEMAS CON TIPOS ENUMERADOS</b>	<b>339</b>	
<b>SECCIÓN 8.5</b>	<b>INTRODUCCIÓN AL TIPO DE DATOS DE CONJUNTOS</b>	<b>342</b>	
	Ejercicios de la sección 8.5	346	
<b>SECCIÓN 8.6</b>	<b>TÉCNICAS DE PRUEBA Y DEPURACIÓN</b>	<b>347</b>	
	Compatibilidad de tipos	347	
	Validación de entradas	351	
<i>Recordatorios de Pascal</i>		352	
<b>SECCIÓN 8.7</b>	<b>REPASO DEL CAPÍTULO</b>	<b>353</b>	
<i>Referencias de Pascal</i>		353	
<i>Avance del capítulo 9</i>		355	

<i>Palabras clave del capítulo 8</i>	355
<i>Ejercicios del capítulo 8</i>	355
<i>Problemas del capítulo 8 para resolución en computadora</i>	356
<b>CAPÍTULO 9 ARREGLOS</b>	359
<i>Objetivos</i>	360
<i>Panorama general del capítulo</i>	360
SECCIÓN 9.1 DECLARACIONES DE ARREGLOS	360
Arreglos	360
Procesamiento de arreglos	365
Arreglos paralelos	367
Resumen de procesamiento de arreglos	368
Ejercicios de la sección 9.1	369
SECCIÓN 9.2 BÚSQUEDA Y CLASIFICACIÓN	372
La búsqueda lineal	373
Búsqueda binaria	376
Clasificación	380
Ejercicios de la sección 9.2	384
SECCIÓN 9.3 ARREGLOS MULTIDIMENSIONALES	385
Procesamiento de arreglos bidimensionales	388
Arreglos de tres o más dimensiones	390
Ejercicios de la sección 9.3	390
SECCIÓN 9.4 RESOLUCIÓN DE PROBLEMAS MEDIANTE ARREGLOS	392
SECCIÓN 9.5 ANÁLISIS DE EFICIENCIA DE ALGORITMOS (OPCIONAL)	396
Notación “O”	397
SECCIÓN 9.6 TÉCNICAS DE PRUEBA Y DEPURACIÓN	399
<i>Recordatorios de Pascal</i>	401
SECCIÓN 9.7 REPASO DEL CAPÍTULO	402
<i>Referencias de Pascal</i>	402
<i>Avance del capítulo 10</i>	403
<i>Palabras clave del capítulo 9</i>	403
<i>Ejercicios del capítulo 9</i>	403
<i>Problemas del capítulo 9 para resolución en computadora</i>	405
<b>CAPÍTULO 10 PROCESAMIENTO DE CADENAS DE CARACTERES</b>	411
<i>Objetivos</i>	412
<i>Panorama general del capítulo</i>	412

<b>SECCIÓN 10.1</b>	<b>CADERAS DE CARACTERES Y ARREGLOS EMPACADOS</b>	413
Arreglos empacados		413
Procedimientos estándar de Pascal: <i>pack</i> y <i>unpack</i>		418
Ejercicios de la sección 10.1		419
<b>SECCIÓN 10.2</b>	<b>PROCESAMIENTO DE CADENAS</b>	421
Concatenación		425
Ejercicios de la sección 10.2		426
<b>SECCIÓN 10.3</b>	<b>RESOLUCIÓN DE PROBLEMAS POR MEDIO DE CADENAS</b>	428
<b>SECCIÓN 10.4</b>	<b>PARÁMETROS CONFORMANTES DE ARREGLOS Y CADENAS (OPCIONAL)</b>	435
Parámetros conformantes de arreglo		435
Operaciones con cadenas		437
Procedimiento <i>copcad</i>		439
Procedimiento <i>cadnul</i>		439
Función <i>loncad</i>		439
Procedimiento <i>concat</i>		439
Función <i>compcap</i>		440
Procedimiento <i>subcad</i>		440
Función <i>índice</i>		440
Función <i>Verif</i>		440
Función <i>leelín</i>		440
Procedimiento <i>Poncad</i>		441
Problema completo: tabulación de uso de palabras		441
<b>SECCIÓN 10.5</b>	<b>TÉCNICAS DE PRUEBA Y DEPURACIÓN</b>	450
<i>Recordatorios de Pascal</i>		452
<b>SECCIÓN 10.6</b>	<b>REPASO DEL CAPÍTULO</b>	452
<i>Referencias de Pascal</i>		453
<i>Avance del capítulo 11</i>		454
<i>Palabras clave del capítulo 10</i>		454
<i>Ejercicios del capítulo 10</i>		454
<i>Problemas del capítulo 10 para resolución en computadora</i>		456
<b>CAPÍTULO 11 REGISTROS</b>		461
<i>Objetivos</i>		462
<i>Panorama general del capítulo</i>		462
<b>SECCIÓN 11.1</b>	<b>REGISTROS</b>	463
Denominadores de campo		464
Ejercicios de la sección 11.1		467

<b>SECCIÓN 11.2</b>	<b>REGISTROS JERÁRQUICOS Y ARREGLOS DE REGISTROS</b>	469
	Registros jerárquicos	469
	Arreglos de registros	471
	Proposición WITH	472
	Ejercicios de la sección 11.2	475
<b>SECCIÓN 11.3</b>	<b>RESOLUCIÓN DE PROBLEMAS MEDIANTE REGISTROS</b>	478
	Clasificación de un arreglo de registros	478
	Clasificación de burbuja	480
	Resolución del problema	481
<b>SECCIÓN 11.4</b>	<b>REGISTROS VARIANTES (OPCIONAL)</b>	483
<b>SECCIÓN 11.5</b>	<b>TÉCNICAS DE PRUEBA Y DEPURACIÓN</b>	486
	<i>Recordatorios de Pascal</i>	488
<b>SECCIÓN 11.6</b>	<b>REPASO DEL CAPÍTULO</b>	489
	<i>Referencias de Pascal</i>	489
	<i>Avance del capítulo 12</i>	491
	<i>Palabras clave del capítulo 11</i>	492
	<i>Ejercicios del capítulo 11</i>	492
	<i>Problemas del capítulo 11 para resolución en computadora</i>	495
<b>CAPÍTULO 12 ARCHIVOS</b>		499
	<i>Objetivos</i>	500
	<i>Panorama general del capítulo</i>	500
<b>SECCIÓN 12.1</b>	<b>TIPOS DE ARCHIVOS Y VARIABLES</b>	501
	Tipos de archivos	502
	Parámetros de archivos	503
	Almacenamiento temporal para archivos ( <i>buffers</i> )	504
	Procedimientos y funciones estándar para manipulación de archivos	505
	Variables de archivos	508
	Ejercicios de la sección 12.1	509
<b>SECCIÓN 12.2</b>	<b>ARCHIVOS DE TEXTO</b>	510
	Edición de un archivo de texto	513
	Ejercicios de la sección 12.2	516
<b>SECCIÓN 12.3</b>	<b>RESOLUCIÓN DE PROBLEMAS CON ARCHIVOS</b>	517
	Fusión	518
	Resolución del problema	519

<b>SECCIÓN 12.4</b>	<b>TÉCNICAS DE PRUEBA Y DEPURACIÓN</b>	<b>521</b>	<b>xix</b>
	<i>Errores de Eof y Eoln</i>	522	CONTENIDO
<i>Recordatorios de Pascal</i>		523	
<b>SECCIÓN 12.5</b>	<b>REPASO DEL CAPÍTULO</b>	<b>524</b>	
<i>Referencias de Pascal</i>		524	
<i>Avance del capítulo 13</i>		525	
<b>Palabras clave del capítulo 12</b>		525	
<i>Ejercicios del capítulo 12</i>		525	
<i>Problemas del capítulo 12 para resolución en computadora</i>		527	
<b>CAPÍTULO 13 INTRODUCCIÓN A LAS ESTRUCTURAS DE DATOS</b>		<b>533</b>	
<i>Objetivos</i>		534	
<i>Panorama general del capítulo</i>		534	
<b>SECCIÓN 13.1</b>	<b>CONJUNTOS</b>	<b>534</b>	
Operadores de conjuntos		537	
Ejercicios de la sección 13.1		541	
<b>SECCIÓN 13.2</b>	<b>APUNTADORES</b>	<b>543</b>	
Ejercicios de la sección 13.2		547	
<b>SECCIÓN 13.3</b>	<b>LISTAS ENCADENADAS</b>	<b>548</b>	
Búsquedas en listas encadenadas		550	
Inserción en listas encadenadas		551	
Eliminación en listas encadenadas		552	
Ejercicios de la sección 13.3		553	
<b>SECCIÓN 13.4</b>	<b>ÁRBOLES</b>	<b>554</b>	
Búsqueda en árbol binario		555	
Recorridos		557	
Ejercicios de la sección 13.3		559	
<b>SECCIÓN 13.5</b>	<b>PILAS Y COLAS</b>	<b>559</b>	
Ejercicios de la sección 13.5		563	
<b>SECCIÓN 13.6</b>	<b>ABSTRACCIÓN DE DATOS</b>	<b>563</b>	
<b>SECCIÓN 13.7</b>	<b>TÉCNICAS DE PRUEBA Y DEPURACIÓN</b>	<b>564</b>	
<i>Recordatorios de Pascal</i>		565	
<b>SECCIÓN 13.8</b>	<b>REPASO DEL CAPÍTULO</b>	<b>566</b>	
<i>Referencias de Pascal</i>		566	
<i>Palabras clave del capítulo 13</i>		567	
<i>Ejercicios del capítulo 13</i>		567	
<i>Problemas del capítulo 13 para resolución en computadora</i>		569	

A Diagramas de sintaxis	576
B Palabras reservadas de Pascal	583
C Identificadores predeclarados	583
D Funciones y procedimientos estándar	584
E Operadores	585
F Codificación de conjuntos de caracteres	586
G Características diversas de Pascal	587
El procedimiento page	587
La proposición GOTO	587
Procedimientos y funciones como parámetros	588
<i>Glosario</i>	591
<i>Respuestas a los ejercicios</i>	605
<i>Índice</i>	637

# PREFACIO



Este texto se proyectó para un primer curso sobre programación estructurada de computadoras y se centra en las estrategias de resolución de problemas planteadas con un enfoque descendente. Específicamente, se da importancia en todo el libro al análisis de problemas, diseño de algoritmos y resolución por medio de Pascal estándar. El texto es apropiado para varios cursos de introducción a la computación, como:

- Cursos de introducción a las ciencias de la computación que siguen los lineamientos de CS1 (ACM Curriculum '84)
- Cursos de introducción a la programación que emplean Pascal
- La primera mitad de un curso de colocación avanzado sobre ciencias de la computación
- Cursos de lenguajes de programación que emplean Pascal

Además, el texto incluye varias secciones opcionales sobre temas avanzados como son invariantes de ciclo, análisis de eficiencia de algoritmos y parámetros conformantes de arreglos. En todo momento se hacen resaltar las ventajas de programar en Pascal estándar.

## Panorama general

En la primera parte del texto (Caps. 1 a 6), se hace hincapié en la resolución de problemas por medio del diseño descendente y la refinación por pasos. El capítulo 1 es una introducción a la computación que proporciona un panorama general de las estrategias de resolución de problemas que se emplean en todo el libro. Los

capítulos 2 y 3 presentan el lenguaje de programación Pascal para resolver problemas elementales. El capítulo 3 también incluye una sección opcional sobre entrada/salida de archivos de texto para aquellos maestros que motivan a sus alumnos a emplear archivos externos al principio del semestre.

El capítulo 4, que trata del diseño descendente y los procedimientos elementales, es fundamental para el resto del texto. Se explica cuidadosamente la estrategia de diseño descendente para la resolución de problemas junto con la escritura de procedimientos, con mayor atención en el enfoque descendente. Se presentan en forma natural los procedimientos con parámetros y se aplican a problemas específicos. Ésta es, en realidad, una introducción gradual a los parámetros. El tratamiento a fondo de los parámetros se posterga hasta después de que se desarrollen las estructuras fundamentales de control, selección y ciclos. Los capítulos 5 y 6 cubren la selección y los ciclos. El capítulo 6 incluye una sección opcional sobre programas correctos e invariantes de ciclos. También se incluye un resumen de las estructuras de control fundamentales.

El ritmo de los primeros seis capítulos es deliberadamente lento para subrayar las estrategias de resolución de problemas que se emplean en el resto del texto. Después de completar el capítulo 6, el estudiante deberá manejar con facilidad las estructuras de control fundamentales y deberá ser capaz de aplicarlas con la estrategia descendente de resolución de problemas. En la segunda parte del texto, el ritmo es más rápido, ya que existe una gran cantidad de información que es preciso cubrir. Se ponen de relieve las estructuras de datos, pero se siguen incluyendo en cada capítulo las estrategias de resolución de problemas con programas en Pascal completos.

El capítulo 7 incluye un análisis detallado de los procedimientos y las funciones, los parámetros de valor y de variable, y la recursión. El capítulo 8 trata de los tipos de datos, e incluye un análisis de los tipos enumerados y una introducción al tipo de datos de conjunto. El capítulo 9, dedicado a los arreglos, incluye algoritmos de búsqueda y clasificación y una sección opcional sobre análisis de eficiencia de algoritmos. El capítulo 10, que analiza el procesamiento de cadenas de caracteres, incluye una sección opcional importante sobre parámetros conformantes de arreglos. Esta sección contiene un paquete completo de manipulación de cadenas escrito en Pascal estándar. Los capítulos 11 y 12 analizan los tipos de datos estructurados de registro y archivo. El capítulo final es una introducción a las estructuras de datos, y se puede considerar como un capítulo de transición hacia cursos avanzados. Los apéndices incluyen diagramas de sintaxis, palabras reservadas, identificadores estándar, orden de operadores, códigos de conjuntos de caracteres (ASCII y EBCDIC), y diversas características de Pascal, como son la proposición GOTO y el uso de procedimientos y funciones como parámetros. A los apéndices les sigue un glosario de importantes palabras clave.

Todos los capítulos tienen la siguiente estructura general:

- Objetivos
- Panorama general del capítulo
- Contenido del capítulo
- Aplicaciones para solución de problemas
- Técnicas de prueba y depuración
- Recordatorios de Pascal

- Repaso del capítulo
- Referencias de Pascal
- Avance del siguiente capítulo
- Palabras clave
- Ejercicios del capítulo
- Problemas del capítulo para resolución en computadora

Cada capítulo se inicia con una lista de objetivos y un panorama general del contenido. Casi todos los capítulos incluyen una sección sobre resolución de problemas con una aplicación y un programa completo en Pascal. Al final de cada capítulo se encuentra una sección sobre técnicas de prueba y depuración para ayudar al estudiante. En esta sección se incluye una lista de recordatorios importantes de los aspectos de Pascal que se cubrieron en el capítulo. En cada capítulo se incluye un repaso del mismo, un resumen del material de Pascal que se cubrió en él, un avance del siguiente capítulo, una lista de palabras clave, y los ejercicios.

## Ejercicios

Casi todas las secciones de cada capítulo contienen ejercicios que puede realizar el estudiante para autoevaluar su dominio del material. Al final del texto se pueden encontrar las respuestas a todos los ejercicios de las secciones. Los ejercicios del capítulo están situados al final de cada capítulo y se dividen en tres categorías: esenciales, importantes y estimulantes. Los problemas para resolución en computadora se dividen en las mismas categorías, e incluyen especificaciones de entrada/salida e ilustraciones de ejemplos de ejecuciones para que el profesor pueda utilizar los problemas como tareas sin necesidad de agregar detalles. Las respuestas completas a todos los ejercicios de los capítulos y a todos los problemas para resolución en computadora se encuentran en el manual del profesor. Al final del capítulo se puede encontrar una explicación detallada de los ejercicios graduados.

## Resumen de características

Este texto tiene las siguientes características pedagógicas:

- Introducción amplia y completa a la computación con el uso de Pascal
- Estrategia de resolución de problemas con mayor atención en el diseño descendente
- Introducción temprana a los procedimientos y al diseño descendente
- Hincapié en el análisis de problemas y la elaboración de algoritmos
- Muchos ejemplos de programas y aplicaciones en Pascal
- Ejercicios graduados: esenciales, importantes y estimulantes
- Problemas graduados para resolución en computadora con especificaciones explícitas de entrada/salida
- Sección sobre técnicas de prueba y depuración en cada capítulo
- Repaso de cada capítulo, con una sección de referencias de Pascal que resume la sintaxis, semántica y reglas de uso de Pascal

- Recordatorios de Pascal que incluyen listas para probar y depurar en cada capítulo
- Lista de palabras clave para cada capítulo
- Glosario de palabras clave importantes

El texto también incluye el siguiente material opcional

- Introducción temprana a la entrada/salida de archivos
- Exactitud de los programas e invariantes de ciclos
- Parámetros conformantes de arreglos y un conjunto completo de procedimientos y funciones de manipulación de cadenas escritos en Pascal estándar.
- Registros variantes
- Análisis de eficiencia de algoritmos

## **Suplementos APEX**

El Pascal académico con extras (APEX, *Academic Pascal with Extras*) constituye un paquete singular y completo de instrucciones ampliamente detallado para acompañar el presente *Programación con Pascal*. La figura de la página siguiente proporciona un panorama descendente general de los suplementos APEX:

Los suplementos APEX se dividen en dos componentes principales: los suplementos del profesor y los suplementos del estudiante.

### **Suplementos del profesor**

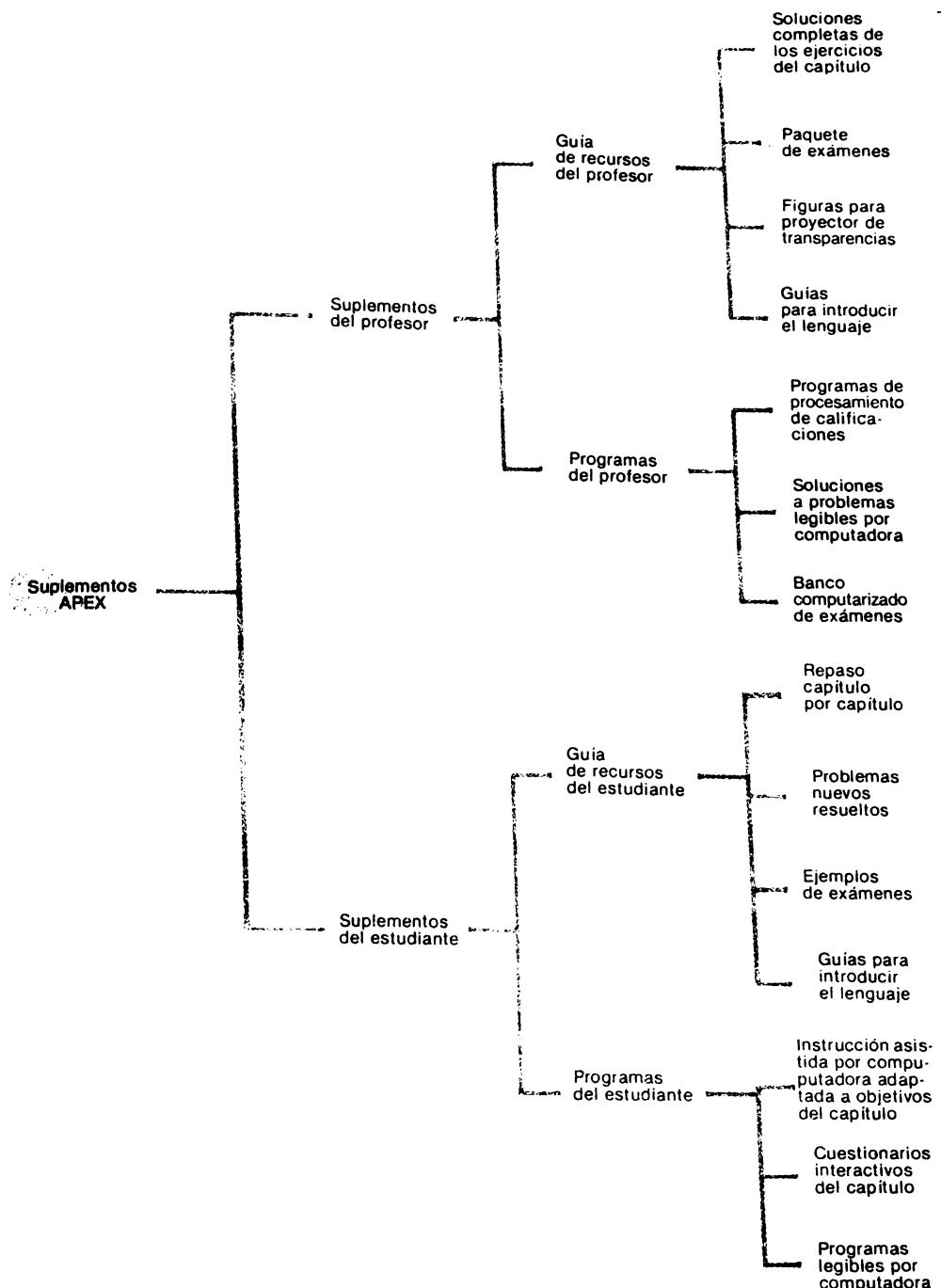
Los suplementos del profesor también se dividen en dos componentes: la *Guía de recursos del profesor* y *Programas del profesor*.

La *Guía de recursos del profesor* contiene los siguientes elementos:

- Soluciones completas a los ejercicios y problemas del capítulo para resolución en computadora
- Sugerencias de enseñanza
- Paquete de exámenes que incluye más de mil preguntas
- Figuras para proyector de derivadas del texto
- Guías de instalación del lenguaje que cubren varias versiones de sistemas no estándar de Pascal (Turbo, UCSD, Apple, etc.)

*Programas del profesor* está disponible para la familia de computadoras Apple II (48 K) e IBM PC (256 K), así como para la mayor parte de los sistemas de computo compatibles, e incluye los siguientes elementos:

- Programas de procesamiento de calificaciones (en Pascal estándar, naturalmente)
- Soluciones que pueden leerse por computadora para todos los problemas del texto



- Banco de exámenes computarizado que genera cuestionarios seleccionados de entre más de mil preguntas

### **Suplementos del estudiante**

Los suplementos del estudiante se dividen en dos componentes principales: la *Guía de recursos del estudiante* y *Programas del estudiante*.

*La Guía de recursos del estudiante* incluye los siguientes elementos:

- Repaso y técnicas de resolución de problemas capítulo por capítulo
- Un gran número de ejercicios nuevos y problemas para resolución en computadora con soluciones completas
- Ejemplos de exámenes que incluyen preguntas para cada capítulo
- Guías para introducir el lenguaje que cubren varias versiones de sistemas de Pascal no estándar (Turbo, UCSD, Apple, etc.)

*Programas del estudiante* está disponible para la familia de computadoras Apple II (48 K) e IBM PC (256 K), así como para la mayor parte de los sistemas compatibles, e incluye lo siguiente:

- Instrucción asistida por computadora y manejada por menús, adaptada a los objetivos de cada capítulo
- Cuestionarios y exámenes para cada capítulo que se pueden resolver en forma interactiva
- Programas completos del texto disponibles en forma legible por computadora

APEX es lo último en sistemas de instrucción sobre computación y proporciona una experiencia de aprendizaje total que utiliza la tecnología de microcomputadoras actual para optimizar el proceso educativo.

## Reconocimientos

Los autores desean agradecer a los siguientes revisores sus valiosos comentarios y sugerencias: Lionel Deimel, Allegheny College; Kalen Delaney, University of California, Berkeley; Terry Gill, Carnegie-Mellon University; Herman Gollwitzer, Drexel University; Bob Holloway, University of Wisconsin-Madison; Layne Hopkins, Mankato State University; Joseph Lambert, Pennsylvania State University; Patricia Murphy, University of Kentucky; Oskars Rieksts, Kutztown University; Richard Rink, Eastern Kentucky University; Henry Shapiro, University of New Mexico; Jill Smudsky, University of Pennsylvania; Mark Stehlík, Carnegie-Mellon University; Ronald Wallace, Blue Mountain Community College, y Jerry Waxman, Queens College.

Los autores desean también agradecer a todos los estudiantes y colegas que contribuyeron en la creación de este texto al leer el manuscrito y resolver los ejercicios. Un agradecimiento particular a Deborah y Rhonda Konvalina, Greg Ostravich, Linda Tuttle, Lori Wernimont y Peggy Wright.

Por último, corresponde un agradecimiento muy especial a Kaye Pace, Shelly Langman y el personal de McGraw-Hill por su impulso, apoyo y dedicación a este proyecto.

*John Konvalina  
Stanley Wileman*

# CAPÍTULO 1

CAPÍTULO 1  
INTRODUCCIÓN  
A LA COMPUTACIÓN

Organización de  
la computadora

Lenguajes  
de computadora

Resolución  
de problemas

Memoria

Instrucciones  
de  
computadora

Lenguajes  
de máquina  
y de bajo  
nivel

Diseño de  
algoritmos

Entrada/  
salida

Unidad  
central de  
procesamiento

Lenguajes  
de alto nivel

Análisis  
de problemas

Resolución en  
computadora

INTRODUCCIÓN  
A LA COMPUTACIÓN

## OBJETIVOS

Después de completar este capítulo, el lector deberá ser capaz de:

- Describir los componentes principales de una computadora: dispositivos de entrada/salida, procesador central y memoria
- Distinguir los diferentes tipos de lenguajes de computación, tales como el lenguaje de máquina, el lenguaje ensamblador y los lenguajes de alto nivel
- Describir los pasos principales en la resolución de problemas por medio de computadoras: análisis del problema, resolución del problema o diseño del algoritmo y resolución en la computadora
- Llevar a cabo y probar la ejecución de los pasos de un algoritmo

## PANORAMA GENERAL DEL CAPÍTULO

Los objetivos de un primer curso sobre computación se pueden resumir como sigue: ayudar al estudiante a desarrollar y aplicar los principios de la computación a la resolución de problemas y a la metodología de programación por medio de un lenguaje de programación. No sólo aprenderá las reglas de un lenguaje de computadora, en este caso Pascal, sino que también elaborará las estrategias que le ayudarán a encontrar soluciones en computadora efectivas para una gran variedad de problemas. Además, el curso proporcionará al estudiante una base firme para estudios y aplicaciones posteriores.

Puesto que la computadora es la herramienta fundamental que constituye la base de la ciencia de la computación, se iniciará este capítulo con una descripción de la organización de la computadora. De manera específica, se analizará la naturaleza de los principales componentes de una computadora: dispositivos de entrada/salida, procesador central y memoria. A continuación, se analizarán las instrucciones de computadora y varios lenguajes de cómputo, así como el lenguaje de máquina. También se explican los detalles de la compilación y ejecución de un programa en lenguaje de alto nivel. Por último, el capítulo se cierra con un análisis general de la resolución de problemas y prueba de programas que es fundamental para el resto del curso. En particular, se hablará del análisis de problemas, resolución de problemas o elaboración de algoritmos y el papel de la computadora en la obtención de soluciones.

## SECCIÓN 1.1 ORGANIZACIÓN DE LA COMPUTADORA

En términos sencillos, una computadora se puede considerar como un procesador de información. Es posible introducir datos e información a la computadora en forma de *entrada* y procesarlos para producir una *salida* (véase la Fig. 1-1).



**Figura 1-1** Modelo sencillo de una computadora.

A los componentes físicos de una computadora, junto con los dispositivos que llevan a cabo la entrada y la salida, se les da el nombre de **hardware**, o equipo. Un **programa de computadora** es un conjunto de instrucciones que ejecuta la computadora. Una parte considerable de este curso trata de la resolución de problemas y la escritura de programas a fin de obtener soluciones de computadora para diversos problemas. Al conjunto de programas que se escriben para una computadora en ocasiones se le da el nombre de **software**.

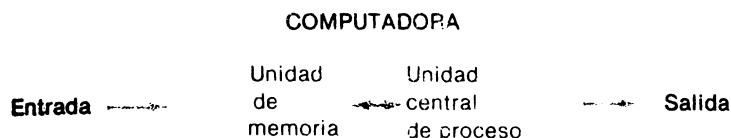
### Entrada/salida

Los dispositivos de entrada/salida (E/S) permiten la comunicación entre el usuario y la computadora. Como ejemplos de dispositivos de entrada se pueden mencionar los teclados, tarjetas perforadas y ratones (dispositivos manuales que sirven para mover un indicador visual en la pantalla). Las impresoras, pantallas de tubo de rayos catódicos (TRC) o terminales de exhibición en video son ejemplos de dispositivos de salida. Las terminales incluyen un teclado y un TRC, por lo que pueden usarse tanto para entrada como para salida. Las instalaciones de cómputo comunes incluyen otros dispositivos de E/S, como son las unidades de disco y de cinta magnética para el almacenamiento y recuperación de información y datos. A los dispositivos de E/S se les conoce también como **dispositivos periféricos**, ya que suelen estar separados de la computadora.

A continuación se depurará el concepto de la computadora como procesador de información y se examinará con más detalle la computadora en sí (véase la Fig. 1-2).

La computadora sin periféricos consta de dos componentes principales, la unidad de memoria y la unidad central de proceso (UCP).

**Figura 1-2** Modelo refinado de una computadora.



## Unidad de memoria

La unidad de memoria está formada por miles de celdas de memoria, a cada una de las cuales se asigna un número único conocido como *dirección de memoria*. Es posible tener acceso a cada celda al hacer referencia a su dirección, lo que permite el almacenamiento y recuperación de información y datos. Además, cuando la computadora ejecuta un programa, éste reside normalmente en la unidad de memoria durante el proceso. La capacidad que tiene la computadora de almacenar un programa en la unidad de memoria y después realizar o *ejecutar* en forma automática las instrucciones es una de las razones principales para utilizar las computadoras en la resolución de problemas.

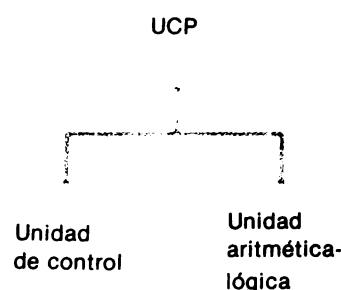
A la unidad de memoria se le llama muchas veces *memoria primaria o principal*. Muchos sistemas de cómputo cuentan con periféricos, como unidades de disco y cinta, que pueden funcionar como dispositivos de almacenamiento secundario y proporcionan así memoria adicional. Ésta, llamada también *almacenamiento secundario*, permite al usuario almacenar permanentemente grandes cantidades de información y datos en discos y cintas.

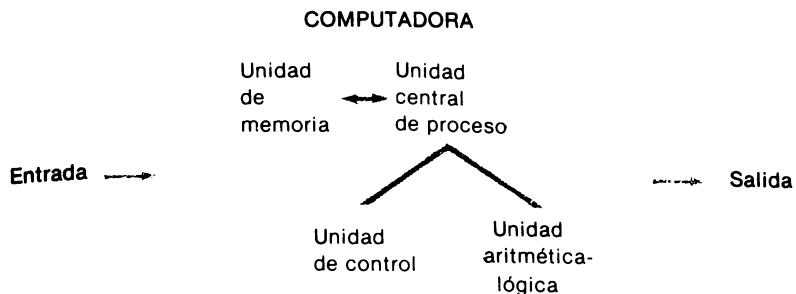
## Unidad central de procesamiento

La UCP dirige y controla el procesamiento de información que lleva a cabo la computadora. Si se observa más de cerca, puede constatarse que la UCP está formada por dos componentes principales: la unidad de control y la unidad aritmética-lógica (véase la Fig. 1-3).

La unidad de control busca y obtiene las instrucciones de los programas que residen en la memoria, interpreta las instrucciones y después dirige su ejecución. La computadora es capaz de repetir este proceso millones de veces cada segundo de manera confiable y precisa. Cuando es preciso realizar cálculos aritméticos como suma, resta, multiplicación o división, la unidad de control ordena a la unidad aritmética-lógica que realice esas tareas. La unidad aritmética-lógica también lleva a cabo las operaciones lógicas, como son comparar dos cantidades o determinar si una condición se cumple o no. Esto proporciona a la unidad de control un mecanismo de toma de decisiones que permite alterar el flujo de una secuencia de instrucciones de computadora.

**Figura 1-3** Unidad central de proceso (UCP).





**Figura 1-4** Estructura de una computadora.

Este modelo depurado de la computadora como procesador de información se ilustra en la figura 1-4.

El conocimiento básico de la organización y estructura de la computadora es una ayuda importante para comprender la forma cómo la computadora ejecuta las instrucciones de un programa, las cuales se escriben normalmente en algún tipo de lenguaje de programación.

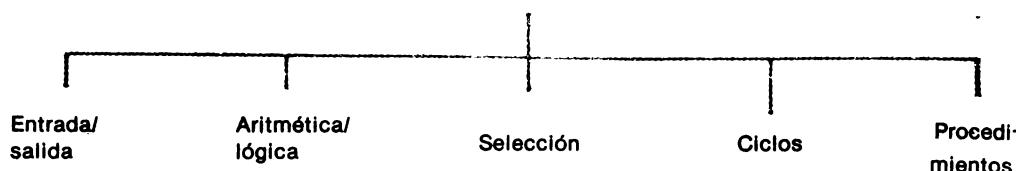
## SECCIÓN 1.2 LENGUAJES DE COMPUTACIÓN

### Instrucciones de computadora

Antes de analizar los lenguajes de computación, conviene hacer un resumen de los tipos fundamentales de instrucciones que puede ejecutar una computadora. Las instrucciones más complejas en varios lenguajes de computación pueden reducirse de hecho a alguno de estos cinco tipos fundamentales (véase la Fig. 1-5).

- *Instrucciones de entrada/salida.* Instrucciones para transferir información y datos entre los periféricos y la memoria principal
- *Instrucciones aritméticas y lógicas.* Instrucciones para realizar operaciones aritméticas (suma, resta, multiplicación, división) o lógicas (resultados que tienen solamente dos valores, falso o verdadero) sobre datos almacenados en la memoria primaria
- *Instrucciones de selección.* Instrucciones que incluyen mecanismos de decisión que permiten al programa elegir diferentes cursos de acción
- *Instrucciones cíclicas.* Instrucciones que permiten repetir una secuencia de instrucciones más de una vez
- *Instrucciones de procedimientos.* Instrucciones que permiten dar nombre a un grupo de instrucciones que constituyen entonces un procedimiento, al cual puede hacerse referencia después por medio de una sola proposición que utilice el nombre del procedimiento

Estas instrucciones se analizarán con mayor detalle en próximos capítulos, sobre todo en lo que toca al lenguaje de computación Pascal.



**Figura 1-5** Instrucciones fundamentales de computadora.

### Lenguaje de máquina

Si se examinara una parte de la memoria de una computadora, se vería que el contenido de cada celda de memoria está representado en términos de *bits*, o dígitos binarios (cero y uno). Por ejemplo, supóngase que el contenido de las celdas de memoria es un programa que comienza en la dirección de memoria 0001 (véase la Fig. 1-6).

Obsérvese que las instrucciones de computadora y cualquier dato que se encuentren en las celdas están representados en código binario (base dos), es decir, en secuencias de bits. En este nivel el programa se expresa en *lenguaje de máquina*. Estas instrucciones en lenguaje de máquina dependen del equipo y el diseño de la computadora digital, por lo que diferentes computadoras pueden tener distintos códigos de máquina para cada instrucción. La programación de una computadora en lenguaje de máquina es un proceso lento y tedioso. Para mejorar esta situación, se crearon lenguajes de computación que permiten escribir programas por medio de proposiciones similares al inglés. Se han creado programas ensambladores y compiladores para traducir estos programas al lenguaje de máquina verdadero.

**Figura 1-6** Celdas de memoria e instrucciones de computadora.

MEMORIA			Instrucciones de computadora en binario.
Celdas de memoria	DIRECCIÓN	CONTENIDO	
	0001	10011001	
	0002	10111001	
	0003	00010101	
	0004	10111010	
	0005	11110000	
	0006	11001010	
...	.....	.....	
...	.....	.....	
	....	....	

CÓDIGO DE LENGUAJE DE MÁQUINA Y NEMONICOS	
1100	ADD
1101	SUB
1110	MPY
1111	DIV

**Figura 1-7** Código de lenguaje de máquina y nemónicos.

### Lenguajes de bajo nivel

Dado que la programación en lenguaje de máquina es difícil, se han elaborado *lenguajes de bajo nivel* para simplificar el proceso. Estos lenguajes casi siempre dependen de la máquina, es decir, dependen del conjunto de instrucciones de la computadora específica de que se trate. Un ejemplo de lenguaje de bajo nivel es el *lenguaje ensamblador*; en él, el programador escribe instrucciones en el nivel de lenguaje de máquina pero utiliza códigos alfabéticos conocidos como *nemónicos*; así, es común expresar los nemónicos para las operaciones aritméticas de suma, resta, multiplicación y división por medio de las palabras ADD, SUB, MPY y DIV, respectivamente.

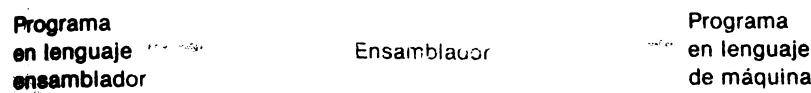
Supóngase que cada operación tiene un código de máquina como se muestra en la figura 1-7. Evidentemente, es mucho más fácil recordar que el nemónico ADD significa adición que recordar que el código de máquina 1100 también significa adición. De hecho, el significado de la palabra *nemónico* es *ayuda para la memoria*.

Después de escribir un programa en lenguaje ensamblador, es preciso traducirlo al lenguaje de máquina para que pueda ejecutarlo la computadora. Casi todos los sistemas de cómputo cuentan con un programa llamado *ensamblador* que traduce un programa en lenguaje ensamblador al programa en lenguaje de máquina correspondiente (véase la Fig. 1-8).

Para simplificar la programación se han ideado otros lenguajes de computadora (conocidos como lenguajes de alto nivel) que son más apropiados para la resolución de problemas en general.

UNIVERSIDAD DE LA REPÚBLICA  
FACULTAD DE INGENIERÍA  
DEPARTAMENTO DE  
DOCUMENTACIÓN Y BIBLIOTECA  
MONTEVIDEO - URUGUAY

**Fig. 1-8** Ensamblador.



## Lenguajes de alto nivel

Como ejemplos de *lenguajes de alto nivel* pueden mencionarse el Ada, BASIC, COBOL, FORTRAN, Lisp, Modula-2, Pascal y PL/1. En los lenguajes de alto nivel las instrucciones de programa o proposiciones se escriben con nombres y palabras comunes que representan los datos que se van a manipular y las acciones que se van a llevar a cabo. Además, es posible escribir los programas en lenguajes de alto nivel de tal forma que son independientes de la máquina; es decir, las proposiciones del programa no dependen del diseño o equipo de una computadora específica. Por tanto, los programas en lenguajes de alto nivel se pueden traducir al lenguaje de máquina y ejecutarse en diferentes computadoras. El programa que lleva a cabo esta traducción se llama *compilador*, y los programas escritos en lenguajes de alto nivel se denominan *programas fuente*. El compilador traduce el programa fuente a un programa llamado *programa objeto*, que es el que se utiliza en la fase de ejecución del programa (véase la Fig. 1-9).

Otra forma de ejecutar un programa escrito en un lenguaje de alto nivel es traducirlo a un programa objeto para una computadora hipotética. A continuación se *interpreta* este programa objeto por medio de un programa (que se ejecuta en una computadora real) que simula a la computadora hipotética.

### Pascal

En este curso se utilizará el lenguaje de alto nivel Pascal para la resolución en computadora de los problemas. El lenguaje de programación Pascal (bautizado en honor de Blas Pascal, matemático francés del siglo XVII) se debe a Niklaus Wirth, quien lo elaboró a finales de la década de 1960 como lenguaje de enseñanza. Por medio del lenguaje Pascal es posible demostrar los principios de la computación y conceptos de programación junto con la resolución de problemas en una forma muy organizada y altamente estructurada.

Cuando se compila una proposición en Pascal, generalmente se traduce a un gran número de instrucciones en lenguaje de máquina. Por fortuna, no es necesario preocuparse de las instrucciones en lenguaje de máquina cuando se programa en Pascal: el compilador se encargará de la traducción. He aquí un ejemplo de programa en Pascal que calcula el área de un triángulo. El programa obtiene la altura y base de un triángulo y a continuación exhibe el área. (Recuérdese que el área de un triángulo es la mitad del producto de la base por la altura.)

**Figura 1-9** Compilador.



```

PROGRAM triángulo (input, output);
VAR altura, base, área : real;
BEGIN
    read (altura, base);
    área := 0.5 * altura * base;
    write (área)
END.

```

En los siguientes dos capítulos se estudiará en detalle la programación en Pascal. De cualquier manera, y aunque el lector no sepa Pascal, obsérvese que las proposiciones del programa son comprensibles y reflejan la solución del problema.

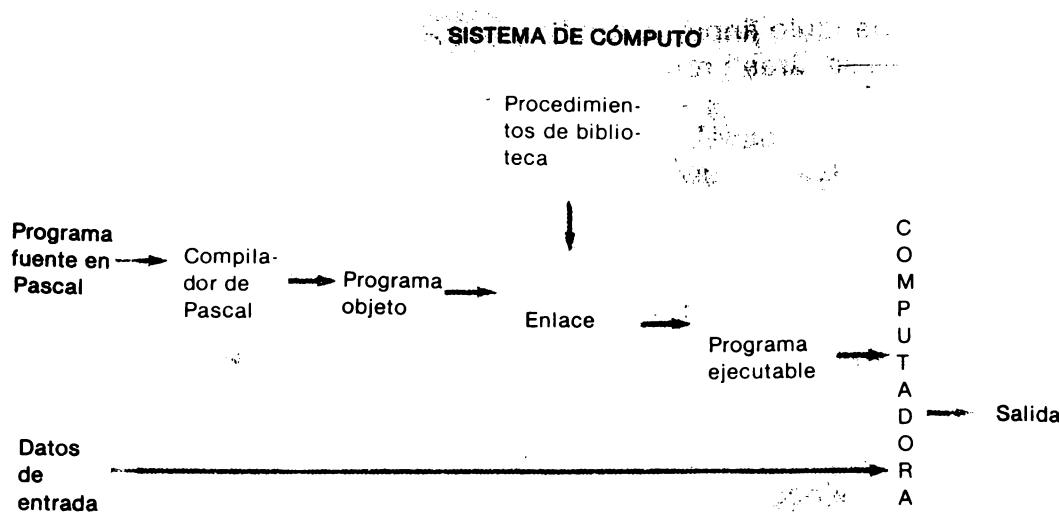
### Compilación y ejecución

El equipo de cómputo, junto con los programas, se denomina *sistema de cómputo*. El proceso de compilar y ejecutar un programa fuente en un lenguaje de alto nivel como Pascal para obtener resultados (salidas), se efectúa dentro del sistema de cómputo. Por lo general, un programa en Pascal obtiene datos de entrada durante la ejecución de un programa. En la figura 1-10 se representa un modelo sencillo de este proceso.

En un sistema de cómputo común, los programas (*software*) del sistema permiten al usuario procesar un programa en Pascal. El *sistema operativo* es un conjunto de programas que se encarga de este proceso. Por ejemplo, para procesar un programa en Pascal, se ordena al sistema operativo que compile el programa fuente para generar el programa objeto. En caso de que no existan errores (llamados *errores de compilación*) en el programa fuente, se ordena al sistema operativo que *enlace* (cargue) el programa objeto con cualquier *procedimiento de biblioteca* (programas de uso frecuente que residen en la biblioteca del sistema) que utilice o al que haga referencia el programa en Pascal. El proceso de enlace da por resultado un *programa ejecutable*. Por último, se ordena al sistema operativo *ejecutar*, o correr, el programa ejecutable con los datos de entrada que se proporcionan. Una vez más, si se supone que no se presentan errores durante la ejecución (llamados *errores de ejecución*), se obtienen las salidas del programa en Pascal. En este momento, el modelo sencillo de compilación y ejecución de un programa en Pascal está representado por la figura 1-11.

**Figura 1-10** Sistema de cómputo.





**Figura 1-11** Compilación y ejecución.

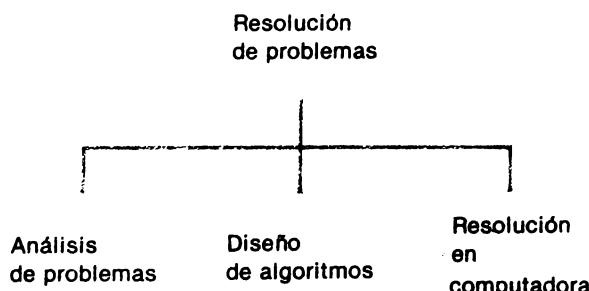
Las instrucciones reales o comandos de sistema para compilar y ejecutar un programa en Pascal varían en los diferentes sistemas operativos. Por ejemplo, algunos sistemas requieren únicamente un comando para procesar el programa en Pascal, mientras que otros pueden requerir comandos separados para compilar, enlazar y ejecutar el programa. Lo que es importante recordar es que una vez que se tienen el programa fuente en Pascal y los datos de entrada, es posible ordenar al sistema operativo que compile y ejecute el programa.

### SECCIÓN 1.3 RESOLUCIÓN DE PROBLEMAS

Cuando se preparan en una computadora las soluciones a los problemas, no basta con conocer las reglas de un lenguaje de programación. La aptitud para resolver problemas es indispensable para programar con éxito.

La resolución de problemas por medio de la computadora se puede dividir en tres pasos principales: análisis del problema, resolución del problema o elaboración del algoritmo, y resolución en la computadora (véase la Fig. 1-12).

**Figura 1-12** Resolución de problemas.





**Figura 1-13** Análisis de problemas.

El primer paso, análisis del problema, requiere una definición clara del problema, el cual debe comprenderse para poderlo analizar con cuidado. ¡No sirve de mucho obtener una solución al problema equivocado! A continuación, es preciso encontrar la solución al problema. Esta solución se puede expresar en términos de un **algoritmo**, que es un procedimiento paso por paso para resolver el problema dado. Por último, para resolver el problema en la computadora, es necesario traducir el algoritmo a un lenguaje de cómputo (en este caso, Pascal) y después verificar o probar que el programa resuelva realmente el problema dado. Ahora se estudiarán los tres pasos con mayor detalle.

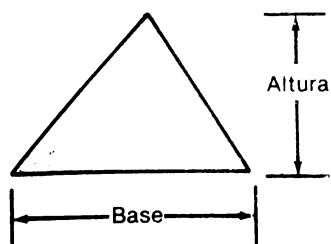
### Análisis del problema

Cuando se especifica un problema que se desea resolver en la computadora, es preciso expresarlo y comprenderlo con gran claridad. El problema se debe definir perfectamente para poder llegar a una solución satisfactoria. Dado que se busca una solución en computadora, es menester describir con el detalle suficiente las especificaciones de entrada y salida del problema. Definir bien el problema y describir con precisión las especificaciones de entrada y salida son requisitos importantes para lograr una solución efectiva y eficiente (véase la Fig. 1-13).

Considérese de nuevo el problema de calcular el área de un triángulo:

*Escríbase un programa de computadora que calcule el área de un triángulo.*

**Figura 1-14** Triángulo con altura y base.



Este problema *no* está bien definido, ya que no se describieron las especificaciones de entrada y salida. Por ejemplo, estas son algunas de las preguntas referentes a las entradas que se han dejado sin respuesta:

- ¿Qué valores se van a introducir?
- ¿Cuántos valores se van a introducir?
- ¿Cuáles son los valores de entrada válidos?

Además, no se han contestado las siguientes preguntas referentes a las salidas:

- ¿Qué valores se van a producir?
- ¿Cuántos valores se van a producir?
- ¿Se deben etiquetar las salidas? y, en caso afirmativo, ¿cómo se hará?
- ¿Cuántas cifras decimales se requieren en los valores de salida?

Estúdiese ahora esta nueva especificación del mismo problema:

### **Problema 1.1**

*Escribase un programa en Pascal al que se le dé la altura y después la base de un triángulo, siendo ambas números decimales positivos. El programa calcula el área del triángulo con los datos de entrada proporcionados. La salida incluye una copia de los datos de entrada y el área con etiquetas apropiadas para cada uno. Los valores de salida deberán imprimirse con dos cifras decimales.*

*Ejemplo de entrada:*

10.5      20.5

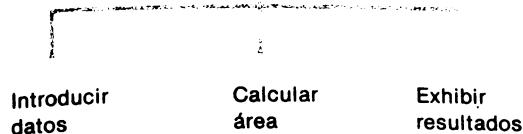
*Ejemplo de salida:*

La altura es 10.50  
 La base es 20.50  
 El área es 215.25

En este momento ya se puede continuar con la resolución del problema o elaboración del algoritmo. Se escogió un problema sencillo para ilustrar el proceso de resolución de problemas y hacer hincapié en la importancia de definir precisamente aun el problema más simple. Es conveniente desarrollar buenos hábitos de resolución de problemas desde el principio del aprendizaje de la escritura de programas para computadora.

### **Diseño de algoritmos**

Los problemas complejos se pueden resolver en forma efectiva con la computadora cuando se descomponen en subproblemas que son más fáciles de resolver que el original. Esta estrategia, llamada en ocasiones *divide y vencerás*, puede



**Figura 1-15** Diseño descendente del problema del área de un triángulo.

aplicarse a la construcción de la solución del problema y a la escritura del programa para computadora. Considérese una vez más el problema de encontrar el área de un triángulo. Este problema se puede dividir en tres subproblemas: 1) introducir los datos, 2) calcular el área y 3) producir los resultados (véase la Fig. 1-15).

Ya se redujo la resolución del problema original a tres subproblemas más sencillos. A continuación se refinará cada subproblema para obtener una solución más detallada.

subproblema	refinación
Introducir datos	Introducir la altura y base del triángulo.
Calcular área	Área = $\frac{1}{2}$ (altura) (base).
Producir resultados	Exhibir altura, base y área.

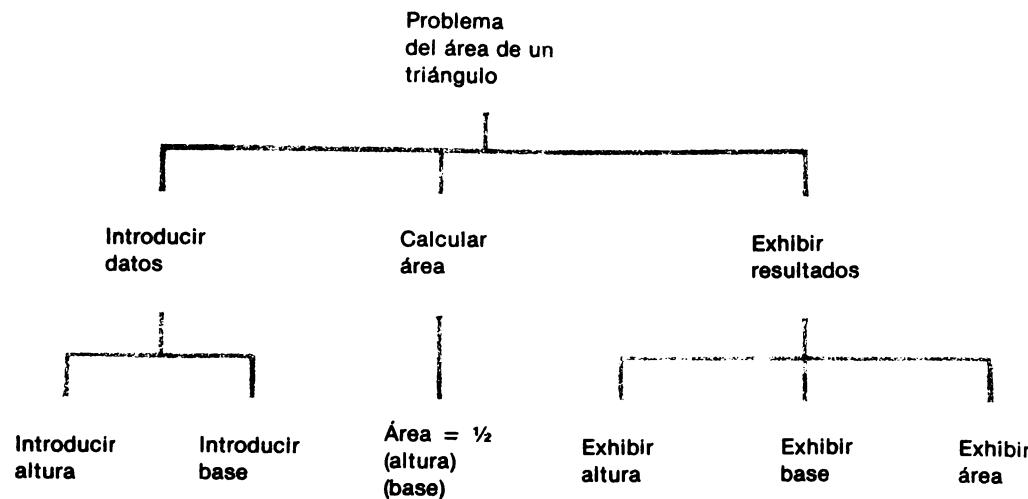
Ahora se tienen los detalles suficientes para producir una solución en la computadora. Este enfoque de dividir un problema en subproblemas y después dividir a su vez los subproblemas hasta que se puedan resolver en la computadora, se denomina *diseño descendente*. El proceso de descomponer el problema en cada etapa se llama *refinación por pasos*. Aunque el problema del triángulo es un caso muy sencillo, el poder y efectividad del diseño descendente se harán más evidentes cuando se ataquen problemas más complejos. En este punto, es muy recomendable que el estudiante aplique el diseño descendente a todos los problemas para resolución en computadora. Así, la transición a problemas complejos será mucho más fácil.

Algunas de las ventajas del diseño descendente son:

- El problema se comprende con facilidad y se organiza lógicamente en partes llamadas **módulos**. Así la solución del problema tiene una estructura.
- Es fácil modificar los módulos.
- Se facilita la prueba de la solución.

Conforme se estudien problemas más complicados y sus soluciones, las ventajas se irán haciendo más obvias.

Ahora se regresará al problema del área de un triángulo y se estudiará el diseño descendente representado en la figura 1-16 (llamado *diagrama de árbol*).



**Figura 1-16** Refinación del problema del área de un triángulo.

El diseño descendente es un bosquejo de la solución del problema. La solución debe traducirse al lenguaje de programación. Obsérvese que en el diseño descendente se parte de una descripción general del problema y se refina en cada nivel hasta que la solución tiene el detalle suficiente para traducirla a un lenguaje de computadora. Este proceso se llama también **diseño de algoritmos**, ya que realmente se está diseñando la solución del problema, o algoritmo. Antes de considerar la solución en computadora del problema, se analizarán los algoritmos con mayor detalle.

### Algoritmos

Un **algoritmo** es un procedimiento paso por paso para resolver un problema dado. Las instrucciones deben ser claras y sin ambigüedades, y lo bastante específicas como para ejecutarse y terminarse en un número finito de pasos. La ejecución de un algoritmo deberá ser susceptible de reducción a una tarea rutinaria. Es por esto que los algoritmos facilitan la escritura de las instrucciones reales para la computadora.

Por ejemplo, es posible expresar así un algoritmo para resolver el problema del área de un triángulo:

#### *Algoritmo de área de un triángulo*

- PASO 1      Introducir la altura y la base.
- PASO 2      Calcular el área =  $\frac{1}{2}$  (altura) (base).
- PASO 3      Exhibir la altura, la base y el área.

Este algoritmo es lo bastante específico como para traducirlo a un lenguaje de cómputo como Pascal. Nótese que los pasos tienen un cierto orden. Así, el paso 1 se ejecuta primero, después el paso 2 y finalmente el paso 3. Una vez escrito el programa para computadora, ésta puede ejecutar las instrucciones en orden. Obsérvese además que este algoritmo contiene el mismo grado de detalle que el nivel más bajo del diagrama de árbol del diseño descendente (Fig. 1-16).

Para hacer una comparación entre las instrucciones del algoritmo y las proposiciones de un lenguaje de alto nivel, la siguiente tabla lista los tres pasos del algoritmo necesarios para obtener el área de un triángulo y las proposiciones correspondientes en Pascal. En los siguientes dos capítulos se estudiarán los detalles de la escritura de un programa completo en Pascal.

<i>paso</i>	<i>algoritmo</i>	<i>proposición en Pascal</i>
PASO 1	Introducir la altura y la base.	read (altura, base);
PASO 2	Calcular el área = $\frac{1}{2}$ (altura) (base).	área := 0.5 * altura * base;
PASO 3	Exhibir la altura, la base y el área.	write (altura, base, área);

## Ejecución de algoritmos

La ejecución de un programa para computadora que representa a un algoritmo dado la realiza la computadora. No obstante, cuando existen errores en el programa o hasta en el algoritmo, es posible que el programador tenga que ejecutar a mano el algoritmo. He aquí un algoritmo sencillo que se encuentra en varios cuadernos de acertijos:

### *Algoritmo de cuaderno de acertijos*

- PASO 1      Pensar en un número.
- PASO 2      Sumar tres al número.
- PASO 3      Multiplicar el resultado del paso 2 por el número dos.
- PASO 4      Restar cuatro al resultado del paso 3.
- PASO 5      Dividir el resultado del paso 4 entre el número dos.
- PASO 6      Restar el número original del paso 1 al resultado del paso 5.
- PASO 7      Escribir el resultado del paso 6.

La ejecución a mano de este algoritmo sería así:

<i>paso</i>	<i>instrucción</i>	<i>resultado de la ejecución</i>
PASO 1	Pensar en un número.	5
PASO 2	Sumarle tres.	8
PASO 3	Multiplicarlo por dos.	16
PASO 4	Restarle cuatro.	12
PASO 5	Dividirlo entre dos.	6
PASO 6	Restarle el número original (5).	1
PASO 7	Escribir el resultado.	se imprime 1

Como ejercicio, ejecútese el algoritmo a mano con distintos números iniciales. ¿Cuál es el resultado? ¿Puede explicar el lector por qué la respuesta siempre es 1? (Véase el ejercicio 1.) Al final del capítulo se pueden encontrar más ejercicios sobre ejecución de algoritmos. Más adelante en el libro se efectuarán algoritmos con instrucciones del lenguaje de programación Pascal.

El algoritmo anterior puede simplificarse considerablemente cuando se utiliza un lenguaje de alto nivel como Pascal. Sin embargo, en vez de decirle que piense en un número, se le dirá a la computadora que obtenga un número de la entrada. Los pasos pueden condensarse así:

PASO 1 Introducir un número.

PASO 2 Calcular el resultado =  $[2(\text{número} + 3) - 4]/2$  – número.

PASO 3 Exhibir el resultado.

Cuando se escribe de esta forma un algoritmo, es decir, en una mezcla de español e instrucciones de computadora, se dice que el algoritmo está escrito en **seudocódigo**. Una vez que se tiene el seudocódigo, casi siempre es una tarea sencilla traducirlo a un lenguaje de cómputo como Pascal. Por ejemplo, los siguientes son el seudocódigo del algoritmo y las proposiciones correspondientes en Pascal:

<i>paso</i>	<i>seudocódigo</i>	<i>proposición en Pascal</i>
PASO 1	Introducir un número.	read (número);
PASO 2	Calcular el resultado = $[2(\text{número} + 3) - 4]/2$ – número.	resultado := $(2 * (\text{número} + 3) - 4) \text{ DIV } 2 - \text{número};$
PASO 3	Exhibir el resultado.	write (resultado);

En la fase de elaboración del algoritmo, una vez analizado cuidadosamente el problema, se utiliza el diseño descendente para dividir el problema en subproblemas (módulos) y organizar la solución. Las refinaciones por pasos ayudan a obtener soluciones detalladas de los subproblemas y algoritmos escritos en seudocódigo, que facilitan el proceso de traducción a un lenguaje de computadora (véase la Fig. 1-17).

A continuación se aplicará el proceso de diseño de algoritmos a un problema más difícil.

**Figura 1-17** Diseño de algoritmos.



*La gerencia de Apex Electronics Company quiere determinar la mediana de los salarios anuales de sus empleados para proporcionar información a un estudio del gobierno. Escríbese un programa que exhiba el número de empleados y la mediana de sus salarios.*

La **mediana** de un conjunto de números es el número que queda en el punto medio cuando se ordenan los números en forma ascendente (de menor a mayor). Por ejemplo, para determinar la mediana de los siguientes números

85, 53, 98, 86, 42

se arreglan primero en orden ascendente (es decir, se *clasifican*)

42, 53, 85, 86, 98

↑  
Mediana

y después se elige el número que queda a la mitad, es decir, 85. Así pues, 85 es la mediana del conjunto de números. Si se excluye la mediana, la mitad de los números quedan arriba de la mediana y la mitad quedan abajo. En este caso el número de datos era non (cinco), por lo que sólo hay un número que queda a la mitad. Sin embargo, si el número de datos es par, existirán dos números en el punto medio y la mediana se definirá como el promedio de esos dos números (su suma dividida entre dos). Por ejemplo, para hallar la mediana de estos seis datos:

62, 86, 97, 84, 98, 56

es preciso clasificar primero los números en orden ascendente:

56, 62, 84, 86, 97, 98

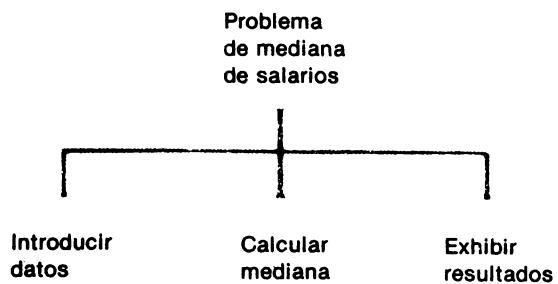
↑ ↑  
Promedio = 85

y después obtener el promedio de los dos números que quedaron a la mitad, 84 y 86, para obtener una mediana de 85. En este punto ya se tiene una definición clara de la mediana de un conjunto de números, por lo que se puede proceder a la resolución del problema.

Si se aplica el proceso de diseño descendente, se puede dividir este problema en tres subproblemas (véase la Fig. 1-18):

- 1 Introducir los datos cuya mediana se desea.
- 2 Calcular la mediana.
- 3 Exhibir el resultado.

Se escogieron problemas relativamente simples para ilustrar mejor el proceso de resolución de problemas y mostrar el grado de precisión que se requiere aun para los problemas más elementales.



**Figura 1-18** Diseño descendente del problema de mediana de salarios.

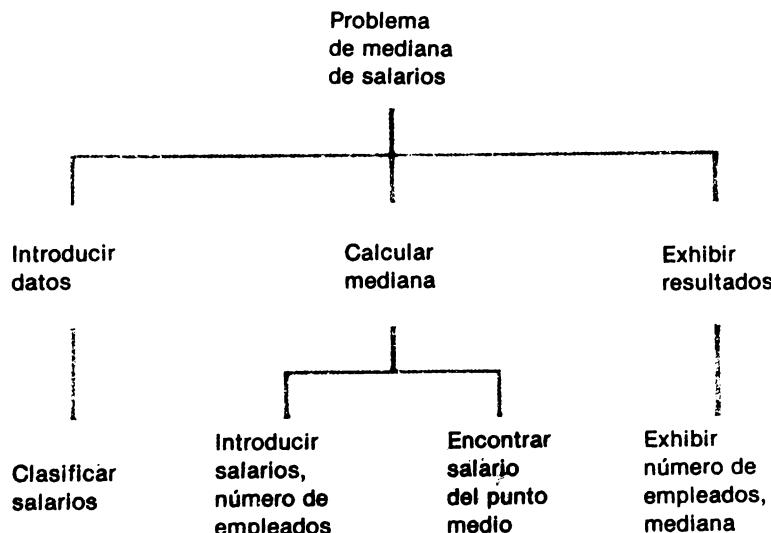
A continuación se refina cada subproblema hasta que se pueda escribir el seudocódigo.

subproblema	refinación
Introducir los datos cuya mediana se desea.	Introducir número de empleados y sus salarios.
Calcular la mediana.	Clasificar los salarios en orden ascendente. Encontrar el salario del punto medio.
Exhibir el resultado.	Exhibir número de empleados y mediana del salario con etiquetas para cada uno.

El diseño descendente se muestra en la figura 1-19.

En este punto es importante comprender que se han evitado los detalles de la resolución de los subproblemas. El uso del diseño descendente permite tener un

**Figura 1-19** Refinación del problema de mediana de salarios.



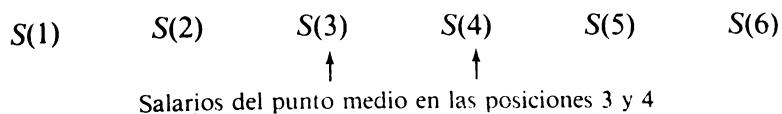
panorama general del problema total y un algoritmo organizado y estructurado para la solución. Ahora se puede proceder a estudiar el algoritmo con mayor detalle.

Si se observa el diseño descendente, se encuentran dos subproblemas que requieren un análisis más profundo: clasificar los salarios y encontrar el salario del punto medio. Los algoritmos de clasificación son muy comunes y muchos sistemas de cómputo cuentan con ellos. En efecto, no es preciso ocuparse de los detalles de la clasificación, ya que se puede utilizar un procedimiento previamente escrito para clasificar los salarios. Más adelante en el libro se aprenderá, de hecho, a escribir un procedimiento de clasificación propio.

Para encontrar el salario del punto medio es preciso saber si el número de empleados es un número par o non. Si hay  $N$  empleados, se representan los salarios de los  $N$  empleados por medio de los símbolos  $S(1)$ ,  $S(2)$ ,  $S(3)$ ,  $S(4)$  y así sucesivamente, hasta  $S(N)$ . Por ejemplo, si fueran cinco los empleados,  $N$  sería cinco, el salario del primer empleado estaría representado por  $S(1)$ , el salario del segundo por  $S(2)$ , etc., hasta el salario del quinto empleado,  $S(5)$ . Si  $N$  es un número non, entonces el salario del punto medio está en la posición  $(N + 1)/2$ . Por ejemplo, con  $N$  igual a cinco, el salario del punto medio está en la posición  $(5 + 1)/2$ , o sea la posición 3:



Si  $N$  es un número par, los dos salarios del punto medio están en las posiciones  $N/2$  y  $(N/2) + 1$ :



La mediana de los salarios es el promedio de estos dos salarios:

$$\frac{S(3) + S(4)}{2}$$

Ya se resolvió el problema de encontrar el salario del punto medio: si  $N$  (el número de empleados) es un número non, la mediana de los salarios estará en la posición  $(N + 1)/2$ ; si no es así,  $N$  deberá ser par, y la mediana de los salarios es el promedio de los salarios en las posiciones  $N/2$  y  $(N/2) + 1$ .

Por fin, se puede escribir el algoritmo en pseudocódigo:

#### *Algoritmo de mediana de salarios*

- PASO 1     Obtener el número de salarios  $N$  de los datos de entrada. Obtener los salarios  $S(1)$ ,  $S(2)$ , ...,  $S(N)$  de los datos de entrada.
- PASO 2     Clasificar los salarios  $S(1)$ ,  $S(2)$ , ...,  $S(N)$  en orden ascendente.

- PASO 3      Calcular la mediana de los salarios.  
 Si  $N$  es non:  
     la mediana está en la posición  $(N + 1)/2$ .  
 Si  $N$  es par:  
     la mediana es el promedio de los salarios en las posiciones  $N/2$  y  $(N/2) + 1$ .  
 PASO 4      Exhibir el valor de  $N$  y de la mediana.

En este momento ya es posible traducir el algoritmo a Pascal. Sin embargo, se requiere una buena cantidad de conocimientos acerca del lenguaje de programación Pascal antes de poder escribir el programa. Por ejemplo, en Pascal es posible determinar si un número es non o par. Si no fuera así, sería necesario refinar aún más el algoritmo.

Una observación importante es que, al resolver los problemas, la mayor parte del tiempo se debe invertir en la fase de definición del problema y en la de diseño del algoritmo. Una vez aprendido un lenguaje de alto nivel como Pascal, el proceso de traducir el algoritmo para ejecución en la computadora se convierte en algo rutinario.

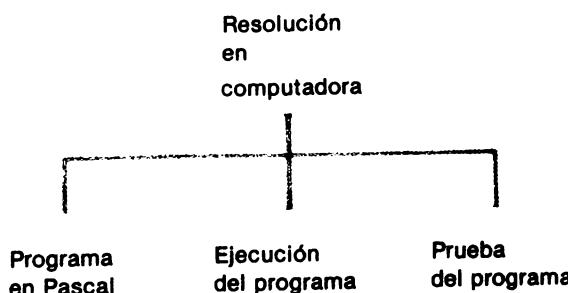
### Resolución por computadora

Cuando se ha diseñado el algoritmo de un problema, se puede proceder con la fase de resolución en la computadora. El primer paso es escribir, o *codificar*, el programa en un lenguaje de programación como Pascal. El siguiente paso es compilar y ejecutar el programa. Este paso incluye la corrección de errores, tanto de compilación como de ejecución. El tercer paso es verificar o probar que el programa sea correcto (véase la Fig. 1-20).

### Programa de Pascal

Después de diseñar el algoritmo para un problema dado, es necesario escribir en detalle las instrucciones para la computadora. Estos detalles se presentarán en los siguientes dos capítulos. Supóngase que se escribió un programa completo en Pascal y se desea continuar con su ejecución en la computadora: es preciso intro-

**Figura 1-20** Resolución en computadora.



ducir en ella el programa. Dos estrategias comunes se conocen como modo interactivo y modo por lotes. En un *sistema interactivo*, el usuario se comunica directamente con el sistema operativo, casi siempre a través de una terminal de video. En un *sistema de procesamiento por lotes*, el usuario entrega al sistema operativo su programa y los datos de entrada que se requieran junto con todos los comandos, para que procese el programa. El usuario debe entonces esperar o volver más tarde por los resultados.

Un sistema interactivo tiene por lo común un programa, llamado *editor de textos*, que permite al usuario crear el programa fuente en Pascal, y cuenta con funciones de edición para modificar el programa. Es usual que el editor de textos cree un *archivo* para almacenar el programa fuente en el sistema de cómputo. Se puede considerar un archivo como una sección del almacenamiento o la memoria a la que se hace referencia por medio de un nombre (creado por el usuario). Si el programa fuente requiere correcciones adicionales, se puede usar el editor de textos para recuperar el programa fuente al especificar el nombre de archivo.

## Compilación y ejecución de programas

Una vez creado el programa en Pascal es posible compilarlo y ejecutarlo. Durante el proceso de compilación, el compilador lee el archivo que contiene el programa fuente y lo traduce a las instrucciones reales en lenguaje de máquina que ejecuta la computadora. El conjunto de instrucciones en lenguaje de máquina se almacena normalmente en un archivo nuevo. Durante el proceso de compilación, el compilador detecta errores (p. ej., el uso incorrecto de Pascal). Estos errores son similares a los errores gramaticales en español y se conocen como errores de *gramática*. Cuando el compilador informa al usuario que descubrió un error de este tipo, el usuario decide cuál es la corrección que hay que hacer al programa fuente, utiliza el editor de textos para corregirlo y vuelve a ordenar la compilación. Si la compilación del programa se lleva a cabo sin problemas, se puede continuar con la ejecución del programa objeto que resulta, el cual queda en una forma susceptible de ser leída por la máquina.

El último paso para procesar el programa es la ejecución del programa objeto. Este paso incluye el procesamiento de todos los datos de entrada que se suministren al programa. Aun cuando el programa se compile correctamente, esto no significa que el programa está libre de errores. Es posible que también se presenten errores durante la fase de ejecución. Por ejemplo, si se intenta dividir entre cero, se exhibirá un mensaje apropiado. Los errores de este tipo por lo general se denominan *errores de ejecución*. Es obvio que hay que corregir esos errores. Esto implica la edición y recompilación del programa fuente y la ejecución del nuevo programa objeto. Los errores de ejecución de los programas se conocen como errores de *semántica*. El proceso de identificar y corregir o eliminar estos errores se llama *depuración*.

Es importante hacer notar que los comandos que se utilizan para compilar y ejecutar los programas de computadora varían en los diferentes sistemas de cómputo. Por ejemplo, es posible que la computadora del lector tenga un *intérprete* de Pascal en vez de un compilador. Cuando se utiliza un intérprete para procesar un programa en Pascal, se traduce y ejecuta cada proposición del programa fuente

antes de procesar la siguiente proposición del programa fuente. Es decir, los intérpretes traducen y ejecutan las proposiciones del programa fuente una por una. Si se encuentra un error, el proceso se detiene y es posible corregir el error inmediatamente. En cambio, los compiladores traducen el programa fuente completo y sólo puede iniciarse la ejecución una vez eliminados todos los errores de sintaxis.

## Prueba de programas

Aun si el programa no tiene errores de compilación o errores de ejecución, es posible que no esté correcto; por ejemplo, puede ser que el resultado sea erróneo. ¡El origen de este error acaso sea el diseño original del algoritmo! A estos errores se les llama a veces *errores de lógica*. Si el programa tiene este tipo de errores, es preciso volver a la fase de diseño del algoritmo y modificarlo, cambiar el programa fuente y compilar y ejecutar una vez más. Para probar que el programa esté correcto, es menester ejecutar varias veces el programa con diversas entradas. Si los resultados son correctos, aumentará la confianza que se puede depositar en el algoritmo y el programa. Sin embargo, esto no garantiza que el programa carezca de errores, ya que podría fallar con otros datos de entrada.

La prueba de los programas para determinar que sean correctos es una parte necesaria de la creación de una exitosa solución en computadora para un problema dado. Conforme se avance en el curso, se presentarán muchas técnicas auxiliares para la prueba de programas. Si se aplican estas técnicas en forma rutinaria, se reducirá la probabilidad de errores y se aumentará la confianza en que los programas son correctos.

## SECCIÓN 1.4 TÉCNICAS DE PRUEBA Y DEPURACIÓN

Al final de cada capítulo el lector encontrará una sección llamada técnicas de prueba y depuración, que le proporcionará consejos útiles para la prueba de programas. A medida que adquiera experiencia en la escritura de programas, el lector deberá aplicar en forma rutinaria muchas de estas técnicas para lograr una solución exitosa a un problema. A continuación se da un ejemplo de prueba de un programa que se puede llevar a cabo durante la fase de diseño de algoritmos del proceso de resolución de problemas. Siempre que se formule un algoritmo para resolver un problema dado, conviene probar el algoritmo con diversos datos de entrada ejecutando los pasos a mano. Los errores de lógica y la posible omisión de pasos en el algoritmo se pueden detectar desde el principio del proceso de resolución de problemas. Si no se prueba el algoritmo, es posible que el programador tenga que pasar muchas horas con la computadora, quizás examinando exhaustivamente las reglas de un lenguaje de programación, en busca de un error que puede remontarse al diseño original del algoritmo. Una ejecución impecable de un algoritmo erróneo no proporcionará los resultados correctos.

Ahora se ejecutarán a mano los pasos del algoritmo de mediana de salarios que se presentó en la sección 1.3. Supóngase que una empresa pequeña tiene cinco empleados. Se probará el algoritmo con los siguientes cinco salarios: \$75 000, \$30 000, \$50 000, \$40 000 y \$60 000. Se eligió un caso de prueba pequeño porque

se va a ejecutar el algoritmo a mano. Los casos de prueba mayores se pueden probar durante la fase de prueba de la solución en computadora.

El primer paso es obtener el número de empleados ( $N$ ) y los salarios. El siguiente paso requiere la clasificación de los salarios en orden ascendente. Por último, se debe calcular la mediana de los salarios y exhibir el resultado. He aquí un resumen de la ejecución a mano del algoritmo para el caso de prueba:

**PASO 1** Obtener el número de empleados ( $N$ ) y sus salarios.

Resultado de la ejecución:

$$N = 5$$

Salarios:	\$75 000	\$30 000	\$50 000	\$40 000	\$60 000
-----------	----------	----------	----------	----------	----------

**PASO 2** Clasificar los salarios en orden ascendente.

Resultado de la ejecución:

Salarios:	\$30 000	\$40 000	\$50 000	\$60 000	\$75 000
-----------	----------	----------	----------	----------	----------

**PASO 3** Calcular la mediana.

Resultado de la ejecución:

Mediana de los salarios: \$50 000

La mediana de los salarios para este caso de prueba fue \$50 000. Nótese que el número de empleados es cinco, que es un número non. Por tanto, la mediana de los salarios está en la posición media (en este caso la tercera) de la lista de salarios clasificados. Como ejercicio, ejecútese a mano el algoritmo de mediana de salarios para una compañía que tenga un número par de empleados (véase el ejercicio 2).

Otra técnica de prueba importante es probar casos excepcionales o especiales. Por ejemplo, supóngase que una empresa tuviera solamente un empleado. ¿Funcionaría el algoritmo de mediana de salarios en este caso especial? Una vez más, como ejercicio, pruébese el algoritmo de mediana de salarios para el caso de un empleado; es decir, pruébese el algoritmo para  $N = 1$  (véase el ejercicio 3). ¿Qué sucedería si los datos de entrada del algoritmo especificaran que *no* hay empleados?

Probar a mano casos pequeños y casos excepcionales es una técnica importante de prueba de algoritmos que ayuda a localizar errores de lógica y pasos faltantes en el algoritmo antes de que se hayan escrito instrucciones en el lenguaje de cómputo.

## SECCIÓN 1.5 REPASO DEL CAPÍTULO

La herramienta fundamental que se utiliza en computación es la computadora. Los componentes principales de una computadora son los dispositivos de entrada/sa-

lida, la memoria y el procesador central. El procesador central consta de dos componentes principales: la unidad de control y la unidad aritmética-lógica.

Existen cinco tipos fundamentales de instrucciones para computadora: de entrada/salida, aritméticas y lógicas, de selección, de ciclos y de procedimientos.

El diseño de una computadora incluye instrucciones de computadora expresadas en código binario en el nivel de hardware. Estas instrucciones se conocen de manera colectiva como lenguaje de máquina. Los lenguajes de bajo nivel, como el lenguaje ensamblador, permiten escribir instrucciones de lenguaje de máquina en forma simbólica por medio de nemáticos. Normalmente, los lenguajes de bajo nivel dependen de la máquina. Los ensambladores son programas que traducen el lenguaje ensamblador a lenguaje de máquina.

Los lenguajes de alto nivel permiten expresar instrucciones de programa o proposiciones de cómputo en expresiones similares a las del inglés. Los compiladores e intérpretes son programas que traducen un programa en lenguaje de alto nivel (programa fuente) a código en lenguaje de máquina (programa objeto). El Pascal es un ejemplo de lenguaje de alto nivel.

El proceso de compilar y ejecutar un programa fuente en un lenguaje de alto nivel incluye traducir el programa fuente en Pascal al programa objeto, enlazar el programa objeto con cualquier procedimiento de biblioteca en caso necesario, para crear un programa ejecutable, y después ejecutar (o “correr”) el programa ejecutable para procesar los datos de entrada.

La resolución de problemas por medio de la computadora se divide en tres pasos principales: análisis del problema, resolución del problema o diseño de algoritmos y resolución en computadora. El análisis del problema requiere expresar clara y precisamente el problema. La solución del problema se expresa en términos de un procedimiento paso por paso llamado algoritmo. El algoritmo se construye por medio de una estrategia de diseño descendente, que divide al problema original en subproblemas más sencillos. Los subproblemas se refinan paso por paso hasta que se pueden convertir en una solución para computadora. Es usual que el algoritmo se escriba en una mezcla de español e instrucciones de computadora, conocida como seudocódigo.

La solución para computadora es el algoritmo escrito en un lenguaje de alto nivel como Pascal. A continuación se compila, ejecuta y prueba el programa. Durante el proceso de compilación, el compilador detecta errores de compilación (errores de sintaxis). Los errores de ejecución se detectan durante el proceso de ejecución. Los errores en el diseño de algoritmos se llaman errores de lógica. El proceso de corregir los errores de un programa se llama depuración.

La prueba de programas es una parte necesaria de la creación exitosa de una solución en computadora para un problema dado. La prueba de casos pequeños a mano y de casos más grandes en la computadora puede reducir el número de errores en un programa y conducir a la resolución eficiente y eficaz del problema.

## Avance del capítulo 2

Si el lector se siente abrumado por el contenido del primer capítulo, no debe desanimarse. El capítulo es básicamente un panorama general del proceso de resolución de problemas por medio de la computadora. Se omitieron muchos detalles

para proporcionar una imagen global de la computadora y su papel en la resolución de problemas. En los siguientes dos capítulos se examinará con más detalle el proceso de resolución de problemas por medio del lenguaje de alto nivel Pascal. Después de escribir unos cuantos programas en Pascal, el lector comprenderá mucho mejor los conceptos presentados en este capítulo.

## Palabras clave del capítulo 1

algoritmo	intérprete
almacenamiento secundario	lenguaje de alto nivel
archivo	lenguaje de bajo nivel
binario	lenguaje ensamblador
bits	lenguaje de máquina
cargar	memoria primaria
código	memoria principal
compilador	nemáticos
depuración	procedimientos de biblioteca
dirección de memoria	procesamiento por lotes
diseño descendente	programa de computadora
dispositivo de entrada	programa ejecutable
dispositivo periférico	programa fuente
dispositivo de salida	programa objeto
divide y vencerás	refinación por pasos
editor	seudocódigo
ejecutar	sintaxis
enlazar	sistema interactivo
ensamblador	sistema operativo
error de compilación	software
error de ejecución	unidad aritmética-lógica
errores de programación	unidad central de proceso
error de lógica	unidad de control
error de semántica	unidad de memoria
hardware	

## INTRODUCCIÓN A LOS EJERCICIOS Y PROBLEMAS PARA RESOLUCIÓN EN COMPUTADORA

Cada uno de los capítulos del texto incluye *ejercicios*, y todos, excepto éste, incluyen *problemas para resolución en computadora*. Los ejercicios permiten repasar el material presentado en el capítulo pero no requieren el uso de una computadora para probar las soluciones. Los problemas para resolución en computadora, aunque quizás sean útiles como problemas de diseño exclusivamente, en realidad están proyectados como expresiones completas de problemas para los cuales se deben desarrollar soluciones (en Pascal). Después de estos ejercicios se encuentra una sinopsis de la forma en que se presentan en subsecuentes capítulos los problemas para resolución en computadora.

Cada ejercicio y problema para resolución en computadora está clasificado. El lector debe intentar resolver siempre los calificados como *esenciales*, ya que reforzán conceptos fundamentales. Los ejercicios *importantes* requieren normalmente más trabajo que los *esenciales*, pero tienen importancia considerable y vale la pena intentar resolverlos. Los ejercicios *estimulantes* son eso precisamente; para obtener la solución es posible que se requiera invertir bastante tiempo. Algunos de éstos podrían ser adecuados para trabajos de un semestre.

## EJERCICIOS DEL CAPÍTULO 1

Los siguientes ejercicios no están relacionados necesariamente con problemas para computadora, pero permiten adquirir práctica para ejecutar a mano los algoritmos. La ejecución a mano de los algoritmos que se escriben es una forma excelente de detectar errores y omisiones.

### ★ EJERCICIOS ESENCIALES

- 1 Ejecútese a mano el algoritmo del cuaderno de acertijos con el número 10. ¿Cuál es el resultado? ¿Puede el lector determinar, matemáticamente, cuál debe ser el resultado del algoritmo?
- 2 Ejecútese a mano el algoritmo de mediana de salarios para el siguiente conjunto de salarios: \$25 000, \$20 000, \$33 400, \$42 500, \$31 000, \$21 000.
- 3 Ejecútese a mano el algoritmo de mediana de salarios para el salario único \$21 000. Obsérvese si la respuesta coincide con la esperada.
- 4 ¿Cuál deberá ser la mediana de los salarios si *no* se proporcionan salarios en los datos de entrada? Modifíquese el algoritmo de mediana de salarios para manejar correctamente este caso.
- 5 Escríbase un algoritmo para producir una solución a un problema que se vaya a resolver en computadora. Utilícese la refinación por pasos para desarrollar el algoritmo y muéstrese todos los pasos que se utilicen en cada nivel y su relación con los pasos que se refinan en el siguiente nivel.

### ★ ★ EJERCICIOS IMPORTANTES

- 6 Supóngase que se tienen cuatro focos dispuestos en un círculo. Los focos se etiquetan con los números 1, 2, 3 y 4 en el sentido de las manecillas del reloj. Supóngase también que se tienen cuatro interruptores conectados de manera que cada uno controla al foco que tiene el número correspondiente. Considérese el siguiente conjunto de instrucciones, sin ejecutar todavía las acciones indicadas.
  - 1) Enciéndase el foco diametralmente opuesto al único que ya está encendido.
  - 2) Si algún foco de número no está encendido, pásese al paso 4.
  - 3) Apáguese el foco de número más bajo y sígase el paso 5.
  - 4) Apáguese el foco de número más alto.

- 5) Apáguese el foco que está junto al foco encendido de número más alto, en la dirección de las manecillas del reloj.
- 6) Apáguese todos los focos de número par que estén encendidos, y terminese.

Ejecútense las instrucciones cuatro veces, cada vez con un solo foco encendido inicialmente. Es decir, enciéndase el foco uno y ejecútense las instrucciones. Después enciéndase únicamente el foco dos y ejecútense las instrucciones. A continuación repítase esto con los focos tres y cuatro. Con base en la experiencia adquirida al ejecutar las instrucciones, ¿qué puede decir el lector acerca del resultado general de ejecutar los pasos?

- 7) Supóngase que se tienen tres recipientes marcados con A, B y C que pueden tener un número arbitrario de canicas de colores. El siguiente conjunto de instrucciones está planeado para ejecutarse a partir del paso 1. Examinense las instrucciones y contéstese después la pregunta que les sigue.
  - 1) Si el recipiente C contiene canicas, pásense todas al recipiente A. En cualquier caso, continúese con el paso 2.
  - 2) Si el recipiente A contiene por lo menos una canica roja, pásese una canica roja del recipiente A al recipiente C. En cualquier caso, continúese con el paso 3.
  - 3) Si el recipiente B contiene por lo menos una canica azul, pásese una canica azul del recipiente B al recipiente A. En cualquier caso, continúese con el paso 4.
  - 4) Si el recipiente B o el recipiente C contienen alguna canica azul, regréssese al paso 2 y continúese a partir de allí. En caso contrario, se ha finalizado la ejecución de las instrucciones.

Ahora supóngase que sólo hay canicas rojas y azules en los tres recipientes. ·  
¿Cuál es el efecto general de ejecutar las instrucciones?

## ESPECIFICACIÓN DE PROBLEMAS PARA RESOLUCIÓN EN COMPUTADORA

Cada uno de los siguientes capítulos incluirá en esta sección problemas que se apegan al siguiente formato. Cada problema es independiente del texto en cuanto a que no se refieren a algoritmos o segmentos de código que aparecen allí. Sin embargo, el maestro puede sugerir modificaciones a los problemas o proporcionar datos específicos para usarlos en la prueba del programa.

- 1 *Las especificaciones de datos de entrada* identifican la forma exacta en la que se proporcionarán los datos de entrada. Si aparece un número variable de datos, se especificarán los límites del número de elementos. En el caso de algu-

- nos problemas se requieren fuentes múltiples de los datos de entrada y se especifica explícitamente la distinción entre estas fuentes.
- 2 *El enunciado del problema* proporciona el objetivo central del problema que se va a resolver. En los casos en que los conocimientos generales no sean suficientes para que el lector produzca un algoritmo, se dará una introducción a la naturaleza del problema.
  - 3 *Las especificaciones de salida* dan la forma exacta en la que se deben exhibir los resultados.
  - 4 Se incluyen *ejemplos de entradas y salidas* para que el lector cuente por lo menos con uno o dos casos sencillos de prueba y para ilustrar los formatos de entrada y salida requeridos.

# CAPÍTULO 2

## CAPÍTULO 2 INTRODUCCIÓN AL PASCAL

Identificadores,  
variables,  
constantes

Tipos de  
datos  
simples

Expresiones  
aritméticas,  
asignaciones

Entero

Real

De  
caracteres

Booleano

# INTRODUCCIÓN AL PASCAL

## OBJETIVOS

Después de completar este capítulo, el lector deberá ser capaz de:

- Reconocer y crear identificadores, constantes y variables en Pascal
- Leer un diagrama de sintaxis
- Declarar variables de tipos simples o estándar: enteras, reales, booleanas y de caracteres
- Reconocer y aplicar la proposición de asignación
- Construir y evaluar expresiones aritméticas con los operadores aritméticos (+, -, \*, /, DIV, MOD) y las funciones estándar (incluidas)
- Introducir, modificar y ejecutar un programa sencillo en Pascal ya existente.

## PANORAMA GENERAL DEL CAPÍTULO

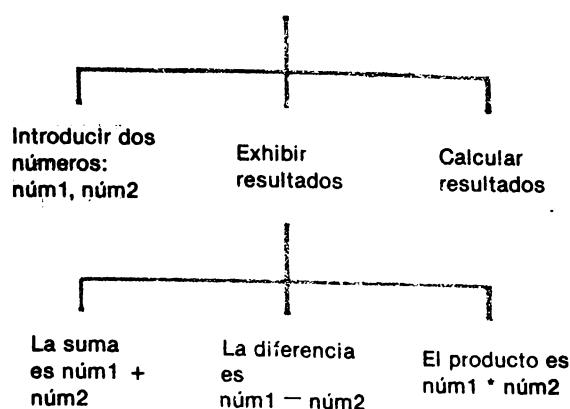
Este capítulo contiene un análisis detallado de los conceptos de programación elementales que hacen uso del lenguaje de alto nivel Pascal. En la primera sección se examinará un ejemplo de programa característico en Pascal para analizar su estructura general. A continuación se explicará con detalle la asignación de nombres a los objetos de Pascal, como son variables y constantes del programa. El Pascal contiene reglas específicas para dar nombre a los objetos y es necesario que el lector conozca estas reglas para escribir programas de computadora. Se presentarán algunos ejemplos de programas completos en Pascal para demostrar el uso de tales reglas.

El Pascal es capaz de almacenar y manipular información de diversos tipos. En este capítulo se examinarán los cuatro tipos de datos simples: enteros, reales, booleanos y de caracteres. Para los datos numéricos en particular, se utilizan los tipos de datos enteros y reales. Se analizará la construcción y evaluación de expresiones aritméticas por medio de los operadores aritméticos de Pascal para la suma (+), resta (-), multiplicación (\*) y división (/, DIV, MOD). Además, el Pascal incluye varias funciones estándar o incluidas. Por ejemplo, la función *sqrt* calcula la raíz cuadrada de un número. Se estudiarán algunas de estas funciones y se dará un ejemplo de su uso.

En la sección de técnicas de prueba y depuración al final del capítulo se presentará un análisis de los errores que surgen en los cálculos aritméticos. Se incluyen ejercicios y problemas para resolución en computadora. Después de completar este capítulo el lector deberá ser capaz de introducir, modificar y ejecutar un programa sencillo en Pascal ya existente. Después de completar el capítulo 3, el lector estará preparado para construir y documentar su propio programa en Pascal.

## SECCIÓN 2.1 EJEMPLO EN PASCAL

En este curso se utiliza el lenguaje de alto nivel conocido como Pascal, cuyo uso se ha extendido por las siguientes razones:



**Figura 2-1** Diseño descendente del programa ejemplo.

- 1 Pascal es un lenguaje de aplicación general que se puede usar para resolver una gama muy amplia de problemas, tanto numéricos como no numéricos.
- 2 Pascal es un lenguaje de programación introductorio bien construido, ideal para usarse en la enseñanza de los conceptos fundamentales de la computación.
- 3 Existen compiladores de Pascal para la mayor parte de los sistemas de cómputo.
- 4 Es posible escribir programas en Pascal en una forma altamente estructurada y organizada, lo que facilita su modificación.

Antes de entrar en los detalles del lenguaje, conviene estudiar la estructura general de un programa en Pascal representativo. Por ejemplo, supóngase que se desea escribir un programa sencillo para leer (o introducir) dos números; calcular la suma, diferencia y producto de este par de números y por último exhibir (imprimir o sacar) estos resultados. El diseño descendente de este programa se muestra en la figura 2-1 y en seguida se presenta el programa en Pascal correspondiente.

---

```

PROGRAM ejemplo (input, output);
(* Programa para obtener dos números de la entrada y *)
(* exhibir su suma, diferencia y producto *)
VAR
  num1, num2, suma, diferencia, producto : integer;
BEGIN
  (* Obtener los dos números *)
  write ('Por favor escriba dos números enteros: ');
  readln (num1, num2);
  (* Calcular suma, diferencia y producto *)
  suma := num1 + num2;
  diferencia := num1 - num2;
  producto := num1 * num2;

```

```
(* Exhibir los resultados *)
writeln ('Los números son ', num1, num2);
writeln ('La suma es      ', suma);
writeln ('La diferencia es ', diferencia);
writeln ('El producto es  ', producto)
END.
```

---

Ahora se examinará el programa línea por línea. El lector no debe preocuparse si no entiende todo desde la primera vez. Más adelante en el capítulo se analizarán estos detalles y más ejemplos.

Considérese la primera línea del programa:

**PROGRAM ejemplo (input, output);**

Esta línea es el encabezado del programa y es necesario en todos los programas en Pascal. El nombre de este programa es *ejemplo*. Las palabras *input* (entrada) y *output* (salida) indican que se están utilizando los dispositivos estándar de entrada y salida definidos en el sistema de cómputo. Si se utiliza un sistema interactivo, es probable que estos dispositivos correspondan a la terminal usada.

Las dos siguientes líneas del programa

**(\* Programa para obtener dos números de la entrada y \*)
(\* exhibir su suma, diferencia y producto \*)**

se llaman *comentarios*. Los comentarios sirven para incluir documentación y otro tipo de información para el programador y demás personas que lean el programa fuente. Es decir, se emplean comentarios para explicar partes del programa a un lector humano. En este caso el comentario define el objetivo del programa *ejemplo*. Los comentarios siempre están encerrados entre los caracteres (\* y \*), o los caracteres [ y ] y, excepto por el hecho de que los incluye en los listados del programa, el compilador de Pascal hace caso omiso de ellos.

Después de este comentario se encuentra:

**VAR num1, num2, suma, diferencia, producto : integer;**

Esto define (o *declara*) las variables (*localidades de memoria* a las que se hace referencia por nombre) que se utilizarán en el programa. En Pascal, todos los datos a los que el programador asigna nombres se deben declarar antes de las proposiciones del programa que los utilizan. En este caso la palabra **VAR** precede a la lista de nombres: *num1, num2, suma, diferencia y producto*. Éstos son los nombres de las variables que se van a utilizar en las proposiciones del programa que siguen. Después de la lista de variables está un signo de dos puntos (:) y la palabra **integer** (entero). Esto indica que las variables precedentes se utilizarán únicamente para guardar valores enteros (es decir, números enteros positivos o negativos).

Después de la declaración de variables están las instrucciones del programa en sí, o *proposiciones ejecutables*. (La declaración de variables no se considera proposición ejecutable, ya que no especifica operación alguna sobre los datos. La diferencia entre las declaraciones y proposiciones ejecutables en Pascal es similar

Las proposiciones ejecutables del programa están encerradas entre las palabras BEGIN y END. El programa *ejemplo* tiene nueve proposiciones (o instrucciones), y están separadas por signos de punto y coma. Obsérvese que la última proposición antes de la palabra END no requiere punto y coma. Adviértase que BEGIN y END no son proposiciones y, por tanto, no están separadas por signos de punto y coma. Cuando se analice una sola proposición en el texto se omitirá el punto y coma que le sigue, ya que técnicamente no es parte de la proposición. Obsérvese también que se incluyen tres comentarios, cada uno de ellos flanqueado por paréntesis y asteriscos: (\* y \*). Ahora se examinarán las proposiciones ejecutables con mayor detalle.

**write ('Por favor escriba dos números enteros: ')**

Esta proposición ordena a la computadora que exhiba el mensaje “Por favor escriba dos números enteros:” en el dispositivo de salida. El objetivo de esta línea es indicar al usuario que introduzca los números que va a procesar el programa.

**readln (núm1, núm2)**

Después de la indicación se ordena a la computadora que obtenga dos números (del dispositivo de entrada) y los almacene en las variables llamadas *núm1* y *núm2*. Si el usuario hubiera escrito 35 y 10, en ese orden, entonces se almacenaría el valor 35 en *núm1* y el valor 10 en *núm2*.

Suma	$\mathbf{:=}$	núm1 + núm2
diferencia	$\mathbf{:=}$	núm1 – núm2
producto	$\mathbf{:=}$	núm1 * núm2

Estas proposiciones, llamadas proposiciones de *asignación*, llevan a cabo la suma, resta y multiplicación, en ese orden. (Nótese que se omitieron los signos de punto y coma para hacer resaltar su función de separadores; *no* se utiliza el punto y coma para terminar proposiciones!) En cada proposición, las variables a la derecha del símbolo  $\mathbf{:=}$  se suman, restan y multiplican. El resultado de cada operación reemplaza entonces (se almacena en, o se *asigna*) al valor de la variable que está a la izquierda del símbolo  $\mathbf{:=}$  y sustituye a cualquier valor que pudiera haber tenido antes. El símbolo  $\mathbf{:=}$ , al igual que los símbolos (\*) y (\*) se forma con dos caracteres distintos sin espacio entre ellos.

Supóngase una vez más que el valor de *núm1* es 35 y el de *núm2* es 10. Después de que la computadora ejecute las tres proposiciones de asignación, los valores de suma, diferencia y producto serán:

Suma	45	(ya que $35 + 10 = 45$ )
Diferencia	25	(ya que $35 - 10 = 25$ )
Producto	350	(ya que $35 \times 10 = 350$ )

**writeln ('Los números son ', núm1, 'y', núm2)**

```
writeln ('La suma es      ', suma)
writeln ('La diferencia es ', diferencia)
writeln ('El producto es  ', producto)
```

Estas últimas cuatro proposiciones ordenan a la computadora que exhiba los resultados en el dispositivo de salida. Cada proposición exhibe un solo renglón. Cuando la computadora ejecuta la primera, por ejemplo, se exhibirá (o se imprimirá) el mensaje “Los números son \_\_\_\_\_,” el valor de *núm1*, la palabra *y* y el valor de *núm2*.

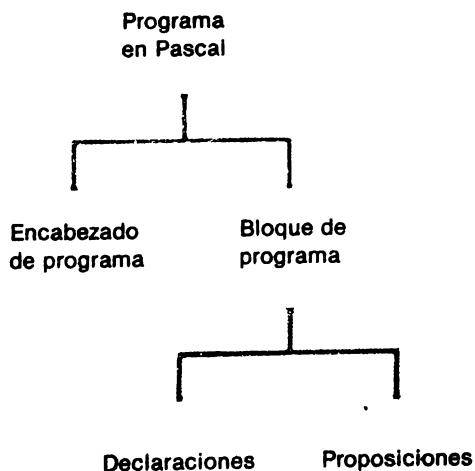
Si este programa se ejecutara en un sistema de cómputo interactivo, se podría observar la siguiente información en la pantalla de la terminal después de completarse la ejecución (los números que escribe el usuario se muestran en **negritas**):

Por favor escriba dos números enteros:	<b>35</b>	<b>10</b>
Los números son	35 y 10	
La suma es	45	
La diferencia es	25	
El producto es	350	

### Estructuras de un programa en Pascal

Un programa en Pascal consta de dos componentes: un encabezado de programa y un bloque de programa. El *encabezado de programa* es una sola proposición que comienza con la palabra PROGRAM. A esta palabra le sigue el nombre que se asigna a todo el programa y una lista entre paréntesis de los nombres asociados a los dispositivos que se utilizarán para entrada y salida. (En Pascal se verán listas en otros lugares; se trata tan sólo de secuencias de nombres u otros elementos separados por comas.) Puesto que el encabezado de programas es una proposición, debe separarse del resto del programa por medio de un signo de punto y coma.

**Figura 2-2** Estructura de un programa en Pascal



El **bloque de programa** es el resto del programa y está formado por dos componentes: las declaraciones y las proposiciones ejecutables. Las **declaraciones** definen (o declaran) los objetos que tienen nombres (a los que se puede hacer referencia) como son las variables y las constantes. Las **proposiciones ejecutables** son las instrucciones que va a llevar a cabo la computadora cuando se ejecute el programa. Estas proposiciones constituyen la parte *activa* del programa, ya que ellas, y sólo ellas, pueden hacer que se realice la manipulación de los datos. El conjunto de proposiciones ejecutables está delimitado por las palabras BEGIN y END. El final de todo el bloque de programa se indica por medio de un punto (que debe seguir naturalmente a la palabra END). La figura 2-2 proporciona un panorama general de la estructura de un programa en Pascal.

En este capítulo se estudiarán en forma detallada conceptos fundamentales de computación como son las variables, constantes, tipos de datos y proposiciones de asignación, así como su relación con el lenguaje Pascal.

## SECCIÓN 2.2 IDENTIFICADORES, CONSTANTES Y VARIABLES

Al escribir programas de computadora en cualquier lenguaje de alto nivel es necesario utilizar nombres para identificar los objetos que se desean manipular. Lo normal es utilizar nombres para designar cosas tales como el programa completo, las variables, algunas constantes, los procedimientos y las funciones. La palabra, o secuencia de caracteres, que forma el nombre de uno de estos objetos se llama **identificador**. En el programa anterior, *ejemplo* es un identificador, como lo son cada uno de los siguientes: *núm1*, *núm2*, *suma*, *diferencia* y *producto*.

Los identificadores se deben crear de manera que cumplan con reglas específicas que pueden variar en los diferentes lenguajes. Las reglas que dictan cómo se combinan los componentes individuales para formar un programa legal se denominan **reglas de sintaxis**. Así, las reglas para formar identificadores legales en Pascal están incluidas en las reglas de sintaxis de Pascal.

### Identificadores en Pascal

Los identificadores que se emplean en los programas en Pascal están formados por un conjunto de caracteres que pueden incluir letras desde la A a la Z y los dígitos del 0 al 9. La única restricción es que el primer carácter de cada identificador debe ser una letra. Muchos sistemas de cómputo tienen dispositivos que incluyen tanto letras mayúsculas como minúsculas, y el Pascal permite utilizar cualquiera de estas cajas. Sin embargo, una letra mayúscula y la minúscula correspondiente son la misma letra para Pascal.

He aquí algunos identificadores legales en Pascal:

NÚM1	H20
núm1 (idéntico a NÚM1)	SS2468
MáxNúm	MÁX
Z2b3004aj19	X

En teoría, los identificadores pueden incluir cualquier número de caracteres. Sin embargo, algunos compiladores de Pascal reconocen únicamente los ocho primeros caracteres de cada identificador. Si se utiliza uno de estos compiladores, debe tenerse cuidado de usar identificadores que estén definidos en forma única por los ocho primeros caracteres, ya que de no hacerlo así se tendrán dos o más identificadores, aparentemente diferentes, que se tratarán como el nombre del mismo objeto. Por ejemplo, para uno de estos compiladores de Pascal *intereses84* e *intereses85* serían el mismo identificador.

Naturalmente, es preferible utilizar identificadores que ayuden a entender qué es lo que está haciendo un programa. Por tanto, conviene resistir la tentación de utilizar identificadores demasiado cortos (como *A*, *B* y *C*) o identificadores que sean “chistosos” (*Niña*, *Pinta* y *Santamaría*) a menos que tengan una relación directa con los objetos a los que nombran. Los identificadores demasiado largos también pueden causar problemas, ya que un error al escribirlos puede dar lugar a dos nombres diferentes cuando se esperaba únicamente uno.

He aquí algunos identificadores ilegales:

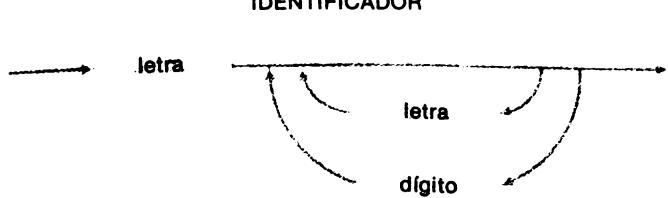
<i>a + b</i>	(El “+” no es ni letra ni dígito.)
<i>costo de la vida</i>	(No se permiten espacios.)
<i>2bornot2b</i>	(No comienza con una letra.)

Algunos identificadores que podrían permitirse como nombres de objetos definidos por el programador están reservados para un uso especial en Pascal. Estos identificadores se llaman *palabras reservadas*. Algunas de las palabras reservadas que se utilizaron en el programa *ejemplo* son PROGRAM, VAR, BEGIN y END. En el apéndice B se puede encontrar una lista completa de estas palabras reservadas. En este texto, las palabras reservadas se escribirán exclusivamente con mayúsculas. Tómese nota, sin embargo, de que esto es únicamente cuestión de tipografía; cuando se escriben palabras reservadas en los programas, estarán formadas por los mismos caracteres que se utilizan en los demás identificadores.

Otro grupo de identificadores que se utilizan en el programa *ejemplo* incluyen *integer*, *input*, *output*, *read* y *writeln*. Éstos se conocen en Pascal como *identificadores estándar*, ya que han sido declarados previamente por el compilador. Es decir, se asocian automáticamente a ciertos objetos aunque no los declare el programa. Si un programa en Pascal sí incluye la declaración de un identificador estándar, la declaración anterior (automática) será cancelada y la última declaración (hecha en el programa) será la única asociación efectiva en el programa. Declarar más de una vez los identificadores estándar casi siempre causa confusión y debe evitarse. En el apéndice C se puede encontrar una lista de los identificadores estándar en Pascal.

## Diagramas de sintaxis

Las reglas de sintaxis de Pascal se pueden representar por medio de ayudas visuales llamadas *diagramas de sintaxis*. El diagrama de sintaxis de la figura 2-3. representa la regla para formar identificadores.



**Figura 2-3** Diagrama de sintaxis de un identificador

Para leer un diagrama de sintaxis se parte del extremo izquierdo y se sigue el “camino” en la dirección que indica la flecha. De inmediato se llega al óvalo marcado “letra”, lo que quiere decir que en este punto se debe escoger una letra. Al continuar después de este óvalo se llega a un “punto de ramificación” similar a la intersección de dos calles. Se puede elegir cualquiera de las tres alternativas (¡nótese que *no* se permiten “vueltas en U”!): continuar derecho hacia la salida, seguir el camino que conduce al óvalo *dígito*, o seguir el camino que pasa por el otro óvalo *letra*. Si se escoge alguna de las dos últimas alternativas, se debe escoger un dígito u otra letra para el identificador, después de lo cual se regresará al punto de ramificación. En seguida se podrán escoger más letras o dígitos, o se podrá salir.

Los diagramas de sintaxis son muy útiles para verificar la validez de los programas en Pascal, ya que proporcionan una representación visual de las reglas para formar programas. Se hablará más sobre los diagramas de sintaxis en una sección posterior. Por ahora, compruebe el lector que ha entendido el diagrama de sintaxis de los identificadores al demostrar que los siguientes identificadores son válidos:

tapa      núm2      a      c22d46

### Constantes

Muchos programas contienen ciertos valores que no deben cambiar durante la ejecución del programa. Estos valores se llaman *constantes*. (Cabe aclarar que algunas variables pueden permanecer sin cambio durante la ejecución de un programa, pero esto se debe únicamente a que las proposiciones ejecutables no hicieron que cambiaron. Las proposiciones ejecutables *nunca* pueden cambiar el valor de una constante.) En Pascal se pueden usar identificadores para dar nombre a las constantes a las que se va a hacer referencia en los programas. Por ejemplo, si se decidiera escribir un programa para calcular el interés de una cuenta de ahorros que paga el 6%, se podría incluir la siguiente declaración:

CONST interés = 0.06;

La palabra CONST está reservada y le sigue el identificador que da nombre a la constante (*interés*), un signo de igual (=, pero no :=) y el valor de la constante (0.06). Las declaraciones de constantes se colocan antes de las declaraciones de variables.

Las declaraciones se separan de otras declaraciones y proposiciones por medio de signos de punto y coma, así como se separan las proposiciones. Dado que las declaraciones de constantes van seguidas por lo menos de otra declaración o proposición, siempre se requerirá un punto y coma al final de la declaración de constantes.

Hay dos razones principales para declarar una constante en un programa en Pascal. Ambas simplifican la programación. En primer término, el programador puede escoger identificadores que reflejen el significado o uso del objeto nombrado en el programa y facilitar así la comprensión de éste. Podrían usarse valores (como 0.06) en varios contextos diferentes en el programa, y el empleo de un identificador para una constante indica cuáles de estos usos son idénticos. Por ejemplo, se podría escribir un programa para determinar el interés de una cuenta de ahorro, así como los impuestos que se deben pagar sobre los intereses. Ambos podrían ser 6%, por lo que se podría usar la constante 0.06 para los dos. Pero si se da un nombre a la tasa de interés (*interés*) y otro a la tasa de impuestos (*impuesto*), el programador (o cualquier persona que lea el programa) podrá distinguir entre los dos usos de la constante.

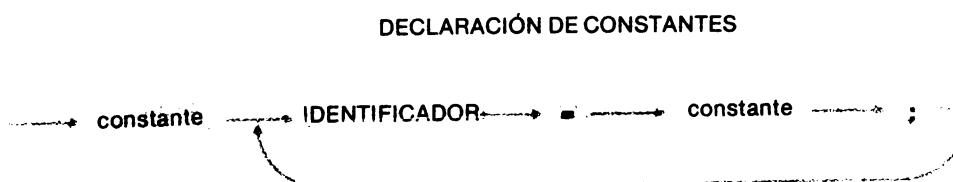
La segunda razón para declarar constantes es contar con una forma sencilla de alterar cada una de las ocurrencias de la constante en el programa si fuera necesario. En el ejemplo de los impuestos, si la tasa a pagar cambiara a 5%, bastaría con cambiar el valor de la constante en la declaración de la tasa de impuestos a 0.05. Así, cada ocurrencia del identificador *impuesto* representaría a la constante modificada. ( Nótese que esto requeriría también una recompilación del programa, ya que es durante la compilación cuando se asocian valores a los identificadores de constantes.)

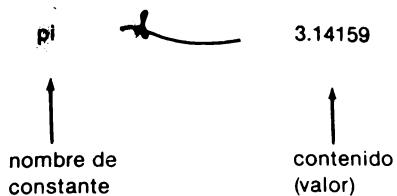
Si se desea declarar más de una constante en un programa, no se repite la palabra reservada CONST, sino que se incluye líneas adicionales de la forma *identificador = constante* separadas por punto y coma:

```
CONST
    interés = 0.06;
    impuesto = 0.05;
```

El diagrama de sintaxis para las declaraciones de constantes se muestra en la figura 2-4. Nótese que en este diagrama de sintaxis se tienen tanto óvalos etiquetados como rectángulos etiquetados. Los caracteres dentro de los óvalos aparecen en forma literal (p. ej., la palabra reservada CONST se debe incluir siempre en una declaración de constantes). Los rectángulos hacen referencia a otros diagramas de sintaxis. En este caso, se deben incluir identificadores y constantes en una

**Figura 2-4** Diagrama de sintaxis de una declaración de constantes.





**Figura 2-5** Almacenamiento de una constante.

declaración de constantes, por lo que se consultarán sus diagramas de sintaxis en los puntos apropiados. El diagrama de sintaxis de las constantes se muestra en el apéndice A. Por el momento puede suponerse simplemente que definen valores del tipo de los que se han estado usando hasta ahora. Para que el lector se convenza de que comprende el diagrama de sintaxis de las declaraciones de constantes, deberá verificar que la siguiente declaración es válida:

#### CONST

tamaño	= 5;
perfecto	= 100.0;
ptoebullición	= 98.6;
iva	= 0.06;

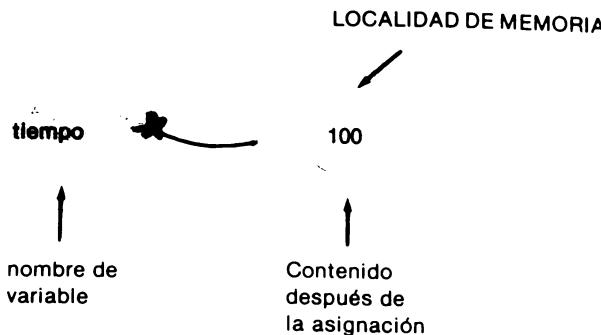
El valor de una constante se almacena en una localidad de la memoria de la computadora. El identificador que se emplea para dar nombre a la constante se refiere a esa localidad de memoria. En este caso el identificador se usa para dar nombre a una localidad de memoria cuyo contenido no podrá cambiar durante la ejecución del programa. Si se llega, por ejemplo, a la siguiente declaración

CONST                pi = 3.14159;

en un programa en Pascal, se reservará una localidad de memoria y se colocará en ella el valor 3.14159. El nombre *pi* quedará asociado a la localidad de tal manera que todas las demás referencias a *pi* en el programa producirán el valor 3.14159. La figura 2-5 ilustra este concepto. El identificador se puede considerar como una etiqueta para la localidad de memoria reservada.

#### Variables

Las *variables* son objetos de un programa que pueden cambiar de valor durante la ejecución. Por fortuna, no cambian de valor arbitrariamente, sino sólo en respuesta a las proposiciones ejecutables del programa. Se utilizan identificadores para dar nombre a las variables en forma similar a como se usan identificadores para dar nombres a las constantes. Al igual que en el caso de las constantes, al declarar una variable se reserva una localidad en la memoria de la computadora y se



**Figura 2-6** Almacenamiento de una variable.

etiqueta con el identificador correspondiente. Pero en el caso de las variables se permite que cambie el contenido de la localidad de memoria (y, por tanto, el valor de la variable) y *no se coloca inicialmente un valor definido en la localidad de memoria*. Esto quiere decir que si se utiliza una variable antes de almacenar un valor en ella (o asignarle un valor) ¡se obtendrá un valor no definido (o inesperado)!

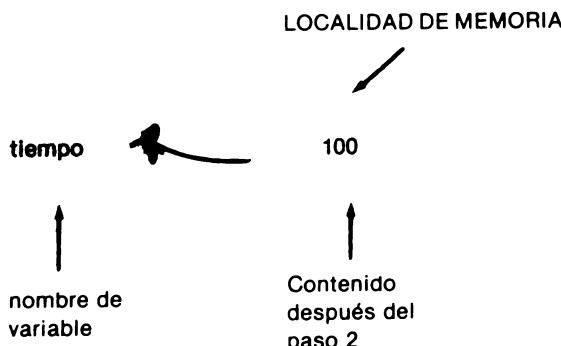
Antes de seguir con el estudio de las variables y su declaración, se examinará la forma cómo se pueden asignar valores a las variables en Pascal. Supóngase que se tiene una variable llamada *tiempo* y se desea que tenga el valor 60. Una forma de lograrlo es emplear una *proposición de asignación* y en este caso la proposición de asignación

`tiempo := 60`

lograría el resultado deseado. Esto se ilustra en la figura 2-6. La proposición de asignación de este párrafo se deberá leer así: “asignar el valor 60 a la variable llamada *tiempo*”.

Es importante observar que las proposiciones de asignación modifican el valor de la variable cuyo nombre aparece a la izquierda del símbolo `:` para que adquiera el valor que está a la derecha. Cualquier valor que haya estado almacenado

**Figura 2-7** Almacenamiento de una variable.



Nombre	Tipo	Valor

**Figura 2-8** Variable en Pascal.

antes en la variable será sustituido por este nuevo valor. Por ejemplo, considérense las siguientes proposiciones:

```
tiempo := 60;
tiempo := 100
```

La primera asigna el valor 60 a la variable *tiempo*. La segunda proposición de asignación sustituye el valor actual de *tiempo* (60) por el valor nuevo, 100 (véase la Fig. 2-7). Sólo existe una localidad de memoria llamada *tiempo*, y sólo puede tener un valor *a la vez*.

Para dar nombre a las variables en Pascal se utiliza un identificador válido. El valor de la variable puede cambiar, pero el nombre no. Una variable en Pascal tiene tres componentes: nombre, tipo y valor (véase la Fig. 2-8).

Para declarar una variable, es preciso especificar el nombre y el tipo de dato que se va a asociar a la variable. El nombre es el identificador que representa a la variable. El tipo de dato indica la forma cómo se va a interpretar el valor de la variable (real, entero, de carácter). El valor de la variable queda determinado por las proposiciones que ejecuta la computadora. Supóngase, por ejemplo, que se desea declarar una variable *costopza*. La variable representará el costo en pesos y centavos de una pieza adquirida. (En realidad, se asociarán pesos y centavos mentalmente a la variable, ya que literalmente la variable sólo va a contener un número.) Puesto que se desea poder representar fracciones decimales con esta variables, se elige el tipo datos *real* para *costopza*. La declaración en Pascal de la variable *costopza* sería<sup>1</sup>

```
VAR costopza : real;
```

En esta declaración se ve la palabra reservada VAR seguida del identificador, símbolo de dos puntos, y el nombre del tipo de datos (en este caso, real).

Es indispensable especificar el tipo de dato de cada variable. En la siguiente sección se estudiarán los llamados tipos de datos simples de Pascal: integer, real, Boolean y char (enteros, reales, booleanos y de caracteres).

<sup>1</sup>Conviene hablar una vez más de los signos de punto y coma. Puesto que las declaraciones de variables siempre irán seguidas de otras declaraciones o proposiciones, se incluye el punto y coma en este ejemplo. Recuérdese, empero, que el signo de punto y coma es un separador.

El concepto de *tipo de datos* es fundamental en computación. Al especificar un tipo de datos particular para una variable se indican las características de los valores que se pueden almacenar en la variable asociada. (Piénsese en la variable como un recipiente para un valor. Entonces el tipo de datos es análogo al tamaño y forma del recipiente.)

El Pascal se considera un lenguaje de *tipos rígidos* porque cada identificador debe tener un tipo asociado a él. Así, el compilador puede verificar durante la compilación que solamente se almacenen valores de los tipos apropiados en las variables. Esto permite detectar algunos problemas antes de ejecutar por primera vez el programa. Por ejemplo, en Pascal no se permite almacenar un número real en una variable entera, y el compilador producirá un mensaje de diagnóstico apropiado. Una analogía aplicable sería examinar un recipiente antes de verter un galón de pintura en él. Si se esperara hasta la *ejecución* y el recipiente resultara demasiado pequeño... Bien, quizás esto sugiera al lector la importancia de detectar el problema antes de la ejecución.

El siguiente ejemplo muestra una declaración de variables que especifica más de una variable:

## VAR

```
tiempo, velocidad : integer;
distancia           : real;
```

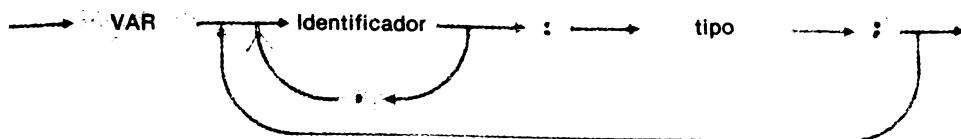
Aquí se ordenó a la computadora que reserve localidades de memoria para dos variables, *tiempo* y *velocidad*, cada una de las cuales es del tipo *integer* (entero), y una localidad de memoria adicional para una variable *distancia*, que es del tipo *real*. Nótese que se permite especificar *tiempo* y *velocidad* en una lista de variables de tipo *integer*. En general, se pueden especificar cuantas variables se desee de esta manera siempre que todas sean del mismo tipo. La variable *distancia* se especificó por separado, ya que su tipo no es el mismo que el de las otras variables.

El tercer componente de una variable, es decir, su valor, se determina durante la ejecución del programa. Ya se vio que las proposiciones de asignación pueden modificar el valor que se asocia a una variable. En la proposición

```
velocidad := 55
```

por ejemplo, el valor 55 se almacena en la localidad de memoria asociada a la variable llamada *velocidad*. Recuérdese que Pascal utiliza el símbolo `:`= para representar la asignación del valor de la derecha (en este caso 55) a la variable cuyo nombre está a la izquierda (*velocidad*). Tómese nota también de que la variable debe tener un valor antes de poderla usar en una expresión. El valor de la variable no estará definido mientras una proposición ejecutable no haga que se coloque un valor en la localidad de memoria reservada para esa variable.

El diagrama de sintaxis de las declaraciones de variables se muestra en la figura 2-9. Puesto que es necesario especificar el tipo de datos de una variable en la declaración, se incluye explícitamente en el diagrama de sintaxis. Una vez más, los rectángulos hacen referencia a otros diagramas de sintaxis. Más adelante se verá con detalle el diagrama de sintaxis de los tipos, pero por ahora se utilizarán



**Figura 2-9** Diagrama de sintaxis de una declaración de variables.

únicamente los tipos simples de Pascal: enteros, reales, booleanos y de caracteres. Con el diagrama de sintaxis el lector deberá ser capaz de verificar que la siguiente declaración es correcta:

```

VAR
  minutos, segundos : integer;
  tasa : real;
  indicador : Boolean;
  calif, final : char;
  
```

### Otro programa en Pascal

A continuación se aplicarán algunos de estos conceptos a un programa real en Pascal llamado *impuesto*. Este programa utiliza el costo de un objeto adquirido como dato de entrada, determina el importe del impuesto sobre venta y finalmente exhibe el precio del artículo y el importe del impuesto. Éste es el programa completo:

---

```

PROGRAM impuesto (input, output);
(*Programa que calcula impuesto de venta del 7% sobre artículos *)
(* adquiridos *)
CONST
  tasaimp = 0.07;
VAR
  costoart, impventa : real;
BEGIN
  (* Obtener costo del artículo de datos de entrada *)
  write ('Por favor escriba el costo del artículo: ');
  readln (costoart);
  (* Calcular impuesto por venta del artículo *)
  impventa := tasaimp * costoart;
  (* Exhibir resultados *)
  writeln ('Costo del artículo: ', costoart:6:2);
  writeln ('Impuesto por venta: ', impventa:6:2);
END.
  
```

---

*Programa impuesto*

## La primera línea del programa

**PROGRAM impuesto (input, output);**

es el encabezado del programa. Contiene, como siempre, la palabra reservada PROGRAM seguida de un identificador que representa el nombre del programa. En este caso se eligió el identificador *impuesto*, ya que sugiere lo que hace el programa. Al nombre le sigue entre paréntesis la lista de dispositivos de entrada y salida que usa el programa.

La siguiente línea del programa contiene únicamente un comentario, encerrado por supuesto entre los símbolos (\* y \*). Puesto que la computadora hace caso omiso de los comentarios, éstos se pueden colocar en el lugar que se desee.

El resto del programa es el bloque de programa que contiene las declaraciones y proposiciones ejecutables. Las declaraciones de constantes y variables se dan precisamente en ese orden:

```
CONST
    tasaimp = 0.07;
VAR
    costoart, impventa : real;
```

En la declaración de constantes, el identificador *tasaimp* da nombre al valor constante 0.07, fracción decimal que representa una tasa de impuesto del 7%. Los identificadores *costoart* e *impventa* dan nombre a las variables que representarán el costo del artículo adquirido y el impuesto por venta de ese artículo, respectivamente. Nótese que cada una de estas variables puede almacenar un número con fracción decimal.

Después de estas declaraciones vienen las proposiciones ejecutables, flanqueadas por las palabras reservadas BEGIN y END. Es útil pensar que estas palabras reservadas siempre constituyen una pareja, como lo hacen aquí, y delimitan otro grupo de proposiciones del Pascal. En este sentido, BEGIN y END son similares a los paréntesis.

En esencia, el programa *impuesto* lee el costo del artículo adquirido y almacena el valor en la variable *costoart*. A continuación, la proposición de asignación

*impventa := tasaimp \* costoart*

multiplica (lo cual se indica en Pascal con el asterisco) *tasaimp* por *costoart* y asigna este valor a la variable *impventa*. Las últimas dos proposiciones hacen que la computadora exhiba el costo original del artículo y el importe calculado del impuesto en el dispositivo de salida. Los detalles de estas proposiciones se explicarán en el siguiente capítulo. Por último, el programa termina con un punto, como todos los programas en Pascal.

El Pascal permite cualquier número de espacios (espacios en blanco y finales de línea y, en algunos sistemas, caracteres de tabulación) entre los identificadores, constantes, palabras reservadas y operadores (como := y \*). Debe aparecer por lo menos un espacio en blanco entre dos identificadores adyacentes (*deotramanerascombinariánparaformarunsoloidentificadorlargo*), pero no es

necesario separar los identificadores de otros símbolos. No obstante, casi siempre es conveniente utilizar espacios para separar los diversos componentes del programa y mejorar así su legibilidad. Considérese por ejemplo el programa que se muestra a continuación. Este programa realizará exactamente el mismo cálculo que el que se mostró antes (el programa *impuesto*). ¿Cuál preferiría leer el lector?

```
PROGRAM impuesto (input, output);(* Programa que calcula impuesto
de venta del 7% sobre artículos adquiridos *)CONST tasaimp = 0.07;VAR
costoart, impventa:real;BEGIN(* Obtener costo del artículo de datos de
entrada *)writeln('Por favor escriba el costo del artículo: '); readln
(costoart);(* Calcular impuesto por venta del artículo *)
impventa := tasaimp * costoart;(* Exhibir resultados *);writeln
('Costo del artículo: ', costoart:6:2); writeln('Impuesto por venta ',
impventa:6:2)END.
```

A continuación se muestra un posible resultado de ejecutar el programa. Los datos que escribe el usuario se muestran en **negritas**.

Por favor escriba el costo del artículo: **30.50**

Costo del artículo: **30.50**

Impuesto por venta: **2.14**

Nótese que el costo del artículo se introduce como 30.50 en vez de \$30.50 (es decir, se omite el signo de dólares). Esto se debe a que la computadora manipula números sin tomar en cuenta sus unidades (en este caso dólares). Esto puede provocar fácilmente errores lógicos en el diseño del programa (como sumar dólares y tasas de impuestos); más tarde se retomará este problema.

Supóngase que el programa tiene muchas proposiciones que utilizan la constante *tasaimp*. Si más tarde se cambiara esta tasa a 8%, bastaría con modificar la declaración de constante así:

```
CONST
    tasaimp = 0.08;
```

en vez de modificar las diversas proposiciones que utilizan *tasaimp*.

## EJERCICIOS DE LA SECCIÓN 2.2

1 Determíñese si los siguientes identificadores son válidos o no.

- |                |            |
|----------------|------------|
| a) Promedio    | b) A1234   |
| c) 1234A       | d) Can Can |
| d) Program     | e) \$XYZ   |
| f) A*b         | g) Jorge   |
| i) 506-74-3981 | j) 4 × 2   |

**2** Explíquese por qué son inválidos los siguientes identificadores.

- |            |                |
|------------|----------------|
| a) 1986    | b) Vel*Tiempo  |
| c) End     | d) 1End        |
| e) Integer | f) Var         |
| g) Sqrt    | h) \$EFFECTIVO |

**3** Determinese si son válidas las siguientes declaraciones de constantes:

- a) CONST impuesto = 0.09; b) CONST pi := 3.14;  
 c) CONST diez = 9; d) CONST Acalif := 90;

**4** Escríbanse declaraciones de constantes para los siguientes valores empleando identificadores alusivos.

- a) El número de días de una semana  
 b) Una persona que pesa 82 kilos  
 c) Una tasa de impuesto por venta del 8%  
 d) Un grupo de 50 estudiantes

**5** Determinese si las siguientes declaraciones de variables son válidas o no.

- a) VAR núm1, núm2 : real;  
 b) VAR núm1; núm2 : integer;  
 c) VAR total, suma, cuenta : integer, real;  
 d) VAR idestudiante, númsegocial : integer;

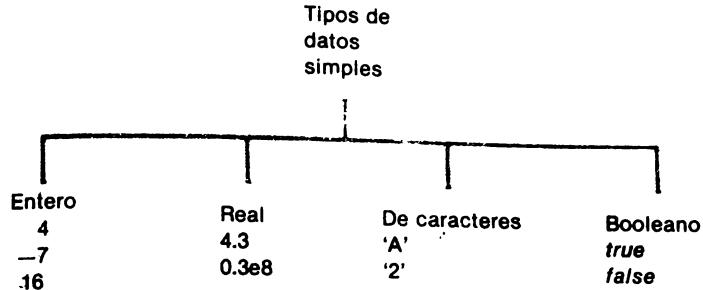
**6** Escríbanse declaraciones de variables para variables que puedan contener los siguientes valores empleando identificadores alusivos.

- a) El número de seguro social del lector  
 b) El año actual  
 c) La tasa de interés sobre un préstamo  
 d) La calificación promedio de varios exámenes

**7** a) Hágase una lista de cinco palabras reservadas que se hayan empleado en los programas vistos hasta ahora.  
 b) Hágase una lista de cinco identificadores estándar que se hayan usado en los programas vistos hasta ahora.

### SECCIÓN 2.3 TIPOS DE DATOS SIMPLES: ENTEROS, REALES, BOOLEANOS Y DE CARACTERES

La computadora es capaz de almacenar y manipular información (datos) de varios tipos. Por ejemplo, el conjunto de calificaciones de exámenes de un curso de computación o de matemáticas es un ejemplo de datos numéricos, ya que lo normal es que cada calificación sea un número entero entre cero y 100. El tipo de datos de las calificaciones se conoce como *entero* (*integer*). Para los saldos de cuentas de cheques en un banco se utiliza un tipo diferente de datos numéricos. Estos



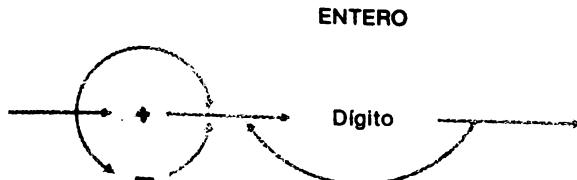
**Figura 2-10** Tipos de datos simples.

saldos se representan normalmente como números enteros con partes fraccionarias (ya que los centavos son fracciones de dólar). Este tipo de datos se conoce como *real*. Los números muy grandes, como los que se utilizan en astronomía, y los muy pequeños, usuales en la física atómica, son también ejemplos de números que pueden ser del tipo *real*. La computadora también es capaz de manipular datos no numéricos como son los nombres de los clientes de un banco. Los programas que procesan palabras, como son los editores de texto, manipulan estos datos no numéricos. El tipo de datos que se usa para representar esta información es el *de caracteres* (en Pascal, este tipo se llama *char*). Por último, se tiene el tipo de datos que se emplea para representar únicamente los valores falso y verdadero, valores que a veces se usan para contestar algunas preguntas de examen. El tipo de datos que tiene únicamente estos dos valores, falso y verdadero, se conoce como *booleano* (término que se deriva de George Boole, pionero de la lógica matemática). La figura 2-10 resume los tipos de datos simples.

Estos cuatro tipos de datos —enteros, reales, de caracteres y booleanos— son los tipos fundamentales y de uso más común, y es por ello que en Pascal se les conoce como tipos de datos *simples* o *estándar*. Los tipos de datos adicionales se analizarán más adelante en el texto.

Los tipos de datos simples describen agrupaciones (o conjuntos) de constantes con diferentes características y las diversas operaciones que se pueden usar para manipular estas constantes. Por ejemplo, la constante 42 es del tipo de dato entero. En la siguiente tabla se muestran algunos ejemplos de constantes de los tipos de datos simples. Se analizará con detalle cada uno de estos tipos.

tipo de dato	descripción	ejemplos	
Entero	Números enteros y negativos, y cero	-30 0	153 96
Real	Números con punto decimal y/o exponente	3.14 1e10	-0.189 6.2e-8
De caracteres	Caracteres como letras, dígitos, signos de puntuación o espacios en blanco	'A' '!' ''	'a' '%' '{'
Booleano	Valores lógicos	verdadero	falso



**Figura 2-11** Diagrama de sintaxis de una constante entera.

### Tipo de datos enteros (`integer`)

El tipo de dato *entero* incluye los números enteros positivos y negativos y el cero. Estos números *no* contienen partes fraccionarias o puntos decimales. Ejemplos de ellos son 32, -16, 0 y 8432. Si no se incluye un signo explícitamente en el número, se supone que es positivo. El diagrama de sintaxis para las constantes enteras se muestra en la figura 2-11.

En Pascal, como en la mayor parte de los lenguajes de programación, no se permiten comas o espacios en blanco dentro de las constantes enteras. Por ejemplo, los siguientes no son enteros válidos:

12 5	(no se permiten espacios entre dígitos)
1,000,000	(no se permiten espacios entre dígitos)
5.0	(no se permite punto decimal)

Verifíquese que éstos no son enteros correctos por medio del diagrama de sintaxis.

Una diferencia importante entre los números que se representan dentro de la computadora y los números "puros" se debe a la memoria finita de la computadora; es decir, el número de localidades de memoria de cualquier computadora, así como el tamaño de cada localidad de memoria, es limitado, por lo que cada computadora tiene un valor entero máximo que se puede representar, ya sea como constante o como variable. Dado que las diferentes computadoras pueden tener diferentes valores máximos, el Pascal tiene una constante entera estándar pre-declarada llamada *maxint*, cuyo valor es el entero más grande que se puede representar en la computadora que se usa. También puede escribirse *-maxint* para representar una aproximación del entero más pequeño que se puede almacenar. (La razón de que *-maxint* sea solamente una aproximación del entero más pequeño no se explicará aquí.) Es posible que *maxint* varíe de una computadora a otra, pero es usual que en las computadoras grandes *maxint* sea igual a  $2^{31}-1$ , o sea 2,147,483,647, mientras que en las computadoras más pequeñas (como las computadoras personales) *maxint* suele ser igual a  $2^{15}-1$ , o sea 32,767.

### Declaración de variables enteras

Supóngase que se desea utilizar una variable llamada *tiempo* para almacenar valores enteros. Antes de usar cualquier variable en Pascal es preciso declararla; pa-

**VAR tiempo : integer;**

Si se desea declarar más de una variable del mismo tipo de datos, se puede utilizar una lista a la izquierda de los dos puntos.

**VAR tiempo, velocidad, distancia : integer;**

Ninguna de estas variables tiene relación con las otras, excepto que son del mismo tipo de datos (entero). (Piénsese en la analogía con tres diferentes botellas de un litro; cada una puede almacenar hasta un litro, pero dado que son diferentes, cada una puede almacenar distintos “valores”.)

### Tipo de datos reales (real)

El tipo de datos *real* incluye números que se pueden expresar con un punto decimal y los números que son demasiado grandes o pequeños para poderse almacenar exactamente como enteros. Si los números reales tienen punto decimal, deben tener por lo menos un dígito antes y otro después del punto decimal. Además, no se pueden escribir con comas entre los dígitos. Considérense los siguientes ejemplos:

forma no válida	forma válida
.25	0.25
-.007	-0.007
5.	5.0
1,000.5	1000.5

Los números muy grandes o muy pequeños se pueden almacenar en variables de Pascal declaradas como reales. Estos números se pueden representar con una notación similar a la notación científica. Para representar en notación científica el número 357.6 se mueve el punto decimal dos posiciones a la izquierda y se multiplica el número por 100, o sea  $10^2$ . La notación científica resultante sería entonces  $3.576 \times 10^2$ . El número 0.000321 también se puede escribir en esta forma si se mueve el punto decimal cuatro posiciones a la derecha y se multiplica por 0.0001, o sea  $10^{-4}$ . El número resultante sería  $3.21 \times 10^{-4}$ .

La siguiente tabla presenta más ejemplos de la representación de números muy grandes o muy pequeños en notación científica. Recuérdese que para escribir un número en notación científica se mueve el punto decimal hasta que se obtiene un número entre uno y diez. A continuación se multiplica ese número por una potencia de diez. Esta potencia (o exponente) de 10 es inicialmente cero, y se incrementa en uno por cada posición que se mueve el punto decimal. Si se mueve el punto decimal a la izquierda, la potencia de 10 aumenta y si se mueve el punto decimal a la derecha, la potencia de 10 disminuye.

número real	notación científica
8569.5	$8.5695 \times 10^3$
100.0	$1.0 \times 10^2$
0.000035	$3.5 \times 10^{-5}$
0.0001	$1.0 \times 10^{-4}$
605.89	$6.0589 \times 10^2$

En Pascal se pueden representar números reales con la *notación de punto flotante*, en la cual la letra *e* (o bien *E*) sustituye a “por 10 elevado a la potencia” (o exponente). Por ejemplo, el número real  $3.576 \times 10^2$  se escribiría en un programa en Pascal  $3.576\text{e}2$ , o  $3.576\text{E}2$ . El número  $3.21 \times 10^{-4}$  se escribiría  $3.21\text{e}-4$ , o  $3.21\text{E}-5$ . El número situado antes de la letra *e* debe ser una constante entera o real, con o sin signo o punto decimal. Si se incluye punto decimal, debe estar precedido y seguido de un dígito, como siempre. El número que va después de la letra *e* es un entero (posiblemente con signo). Este número se llama *exponente*. El diagrama de sintaxis para las constantes del tipo de datos real (Fig. 2-12) presenta la regla en forma más explícita.

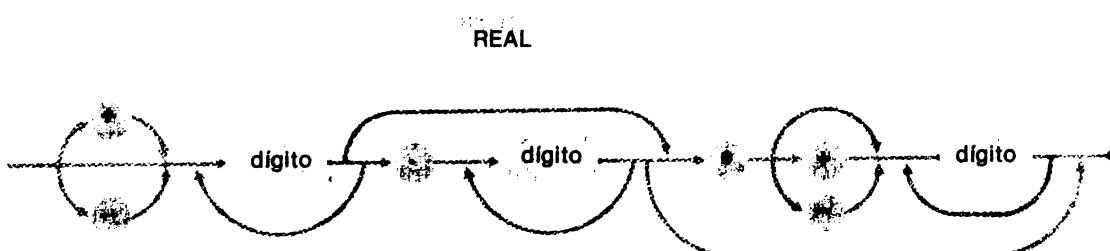
Mediante este diagrama, el lector deberá ser capaz de verificar si las siguientes constantes son válidas o no, según se indica.

CONSTANTES REALES VÁLIDAS		CONSTANTES REALES NO VÁLIDAS	
notación de punto flotante	notación científica	notación de punto flotante	razón
2.6e15	$(2.6 \times 10^{15})$	1.e2	No hay dígito después del punto decimal
-3.2e-6	$(-3.2 \times 10^{-6})$	-6.5e3.6	El exponente no es entero
2e14	$(2 \times 10^{14})$	.2e3	No hay dígito antes del punto decimal
6.5432e01	$(6.5432 \times 10^1)$	3,642.5e-5	No se permiten comas
-4.6e2	$(-4.6 \times 10^2)$		
-4.107e32	$(-4.107 \times 10^{32})$		

### Declaración de variables reales

Puesto que las variables pueden representar muchos valores diferentes, la decisión de utilizar un tipo de datos entero o real es importante. Obviamente, si los valores tienen partes decimales, se debe escoger el tipo real. Puesto que los enteros tienen magnitudes limitadas (de  $-maxint$  a  $maxint$ ), también se debe utilizar

Figura 2-12 Diagrama de sintaxis de una constante real.



el tipo real para representar las cantidades que excedan esta escala. La escala permisible de valores para el tipo de datos real varía en los diferentes sistemas de cómputo.

Dado que el tipo de datos real permite almacenar números con partes fraccionarias y números de mayor escala que los enteros, se podría pensar que los enteros no son necesarios. La respuesta es que los números del tipo de datos real son sólo aproximaciones a valores exactos. La aritmética con números reales puede producir fácilmente respuestas aparentemente incorrectas; los resultados son tan sólo aproximaciones a los valores exactos. Por otro lado, los valores del tipo de datos entero siempre se representan con exactitud, mientras los resultados de operaciones aritméticas no excedan la escala permisible. En la sección de técnicas de prueba y depuración al final del capítulo se dan ejemplos de algunos de los problemas con el tipo de datos real.

La declaración de las tres variables reales *impuesto*, *salario* y *tasa* podría hacerse así:

```
VAR impuesto, salario, tasa : real;
```

### Tipo de datos de caracteres (char)

Los nombres de personas y objetos, palabras, números de pieza y direcciones no son más que secuencias de caracteres. Cada uno de estos caracteres se puede representar por medio de una variable del tipo de datos de carácter, llamado tipo de datos *char* en Pascal. En realidad, los caracteres se representan internamente (en la memoria de la computadora) como códigos enteros pequeños (normalmente dentro de la escala 0 a 127), pero externamente (es decir, en un teclado o impresora), los caracteres son las figuras comunes que se utilizan para representar nombres y direcciones. Las figuras que se manejan generalmente son caracteres alfabéticos (mayúsculas y minúsculas), dígitos decimales, y caracteres especiales y signos de puntuación.

En Pascal las constantes del tipo de datos de carácter se representan por medio de un solo carácter encerrado entre apóstrofos. Los siguientes son ejemplos de constantes del tipo de datos de carácter:

```
'A'      'a'      '6'      ''      '{'      """
```

Aquí la constante 'A' *no* es igual a la constante 'a'. Tampoco es igual la constante de carácter '6' a la constante entera 6. Para representar un espacio en blanco como dato que va a manipular el programa, se utiliza ' ', un espacio encerrado entre apóstrofos. Las llaves también se escriben '{' a fin de que no se confundan con un comentario. Para representar un solo apóstrofo se escribe "" (nótese que *no* se trata de comillas, "'"). Esta forma especial es necesaria porque son los apóstrofos los que delimitan las constantes de carácter.

Las variables de carácter pueden almacenar solamente un carácter. En Pascal se pueden representar constantes de más de un carácter como constantes de cadena de caracteres. Los caracteres aparecen entre apóstrofos. He aquí algunas constantes de cadena de caracteres representativas:

'El impuesto por venta es'  
'Introduzca sus datos, por favor'  
'New York, New York 10020'

Si se desea incluir un apóstrofo en una cadena de caracteres, es necesario escribir dos apóstrofos (*no* comillas). Por ejemplo, las palabras *Alice's Restaurant* se representarían por medio de la constante de cadena de caracteres 'Alice''s Restaurant'.

### Declaración de constantes y variables de caracteres

Las constantes de un solo carácter y las constantes de cadena de caracteres se pueden declarar en la sección CONST del programa y se les nombra igual que las constantes enteras y reales. Considérese este ejemplo:

#### CONST

estrella	= '*' ;
blanco	= ' ' ;
comilla	= " " ;
estado	= 'California' ;
mensaje	= 'Hola, mi nombre es HAL.' ;

Las primeras tres de estas constantes, *estrella*, *blanco* y *comilla*, son constantes de un solo carácter, mientras que las últimas dos, *estado* y *mensaje*, son constantes de cadena de caracteres.

La siguiente es una declaración de variables representativa en Pascal:

VAR letra, grado, inicial : char;

Esta declaración establece variables llamadas *letra*, *grado* e *inicial*, cada una de las cuales se puede utilizar para almacenar exactamente un carácter. Para almacenar varios caracteres se deben utilizar o bien muchas variables (con diferentes nombres) o un arreglo (o lista) de caracteres (que se estudiará un poco más adelante).

### Tipo de datos booleanos

Algunas veces bastan dos valores para representar la información, como sí y no, y verdadero. Por ejemplo, una prueba de falso-verdadero es un examen en el que cada pregunta, se contesta con la respuesta falso o verdadero. Las encuestas muchas veces solicitan una respuesta de sí o no.

El tipo de datos *booleano* se emplea para representar los valores "lógicos" falso y verdadero. Muchas de las pruebas de un programa de computadora darán resultados de este tipo. Por ejemplo, cuando se pregunta si un determinado valor entero es non, la respuesta será falso o verdadero. Existen solamente dos constan-

CONST indicador = true;

Las declaraciones de variables que abarcan al tipo de datos booleano se construyen en forma similar a las de los demás tipos de datos simples. En la declaración

VAR commuta : Boolean;

el identificador *commuta* se usa para hacer referencia a una localidad de memoria que puede almacenar solamente los valores *true* y *false*.

Más tarde se examinará con detenimiento la manipulación de datos booleanos, especialmente en lo que respecta a la realización de pruebas y a la toma de decisiones.

## EJERCICIOS DE LA SECCIÓN 2.3

1 Determíñese cuáles de las siguientes son constantes enteras válidas en Pascal.

- |    |        |    |       |    |        |
|----|--------|----|-------|----|--------|
| a) | 189    | b) | -2.5  | c) | '33'   |
| d) | -55555 | e) | 6,632 | f) | 2.5e03 |
| g) | + 199  | h) | .199. | i) | maxint |

2 Determíñese cuáles de las siguientes son constantes reales válidas en Pascal:

- |    |         |    |         |    |         |
|----|---------|----|---------|----|---------|
| a) | -0.01   | b) | .025    | c) | -3.6    |
| d) | 69.     | e) | 3.6e-06 | f) | 3.e-06  |
| g) | + 8.3e2 | h) | 1.0e1.  | i) | maxreal |

3 Determíñese cuáles de las siguientes son constantes de carácter válidas en Pascal.

- |    |     |    |     |    |       |
|----|-----|----|-----|----|-------|
| a) | A   | b) | 'A' | c) | 'CAT' |
| d) | '8' | e) | '?' | f) | '??'  |

4 Escribábase una constante real en Pascal que represente a cada uno de los siguientes valores.

- |    |                    |    |                       |
|----|--------------------|----|-----------------------|
| a) | $6.21 \times 10^3$ | b) | $0.15 \times 10^{-2}$ |
| c) | $1.66 \times 10^6$ | d) | $22.4 \times 10^{-8}$ |

5 Escribábase el valor de cada una de las siguientes constantes reales de Pascal sin usar exponente.

- |    |         |    |         |
|----|---------|----|---------|
| a) | 2.6e03  | b) | -12e-2  |
| c) | 4.56e10 | d) | -66.5e1 |

**6** Explíquese por qué cada una de las siguientes no es una constante real válida en Pascal.

- |    |       |    |        |    |            |
|----|-------|----|--------|----|------------|
| a) | 10.   | b) | .33    | c) | -2.6e1.5   |
| d) | 5.25- | e) | 2.e-03 | f) | 3.14159... |

**7** Determinese cuáles de las siguientes son constantes válidas en Pascal.

- |    |        |    |     |    |         |
|----|--------|----|-----|----|---------|
| a) | 'e'    | b) | e   | c) | false   |
| d) | 999    | e) | 3.e | f) | -maxint |
| g) | -0.000 | h) | '?' | i) | !       |

## SECCIÓN 2.4 PROPOSICIONES DE ASIGNACIÓN Y EXPRESIONES ARITMÉTICAS

### Asignación

Las proposiciones de asignación como *núm1* := 5, sirven para cambiar el valor almacenado en la localidad de memoria de una variable. Se emplea el símbolo := para indicar que se debe hacer una asignación y se llama *operador de asignación*. Nótese que está formado por dos caracteres separados, y en programas en Pascal se debe escribir sin espacio entre los dos.

El nombre de la variable cuyo valor se va a modificar se debe escribir a la izquierda del operador de asignación. A la derecha se coloca una *expresión* que va a proporcionar el valor nuevo de la variable que se nombra a la izquierda. La figura 2-13 muestra el diagrama de sintaxis de la proposición de asignación. Un poco más adelante se examinarán las expresiones con mayor detalle, pero antes de hacerlo conviene tener en cuenta la importante diferencia entre los símbolos = y := (operador de asignación). Como ya se vio, el signo de igual cuando aparece solo (es decir, sin un símbolo de dos puntos inmediatamente antes) indica que los elementos de ambos lados son *idénticamente iguales*, como en la declaración de una constante. El símbolo de igual también se usa para probar la igualdad de dos valores. Por otro lado, el operador de asignación cambia el valor de la variable que está a su izquierda y trabaja en forma independiente de la relación previa entre los valores que están a ambos lados de él. Por ejemplo, la declaración

CONST velocidad = 55;

**Figura 2-13** Diagrama de sintaxis de una proposición de asignación.



**rapidez := 55**

es una proposición ejecutable que ordena a la computadora cambiar el valor de la variable *rapidez* al nuevo valor entero 55.

La computadora ejecuta las proposiciones de asignación en dos pasos básicos. En el primero se determina el valor de la expresión que está a la derecha del operador de asignación. Este paso puede ser muy complicado (como se verá), pero el resultado final es simplemente un valor de un tipo específico. En el segundo paso se almacena este valor en la localidad de memoria que corresponde a la variable cuyo nombre aparece a la izquierda del operador de asignación, y se sustituye así su valor anterior.

Es posible utilizar el mismo nombre de variables tanto en el lado izquierdo del operador de asignación como en la expresión del lado derecho. Puesto que primero se evalúa la expresión, es el valor actual de la variable el que se emplea en la evaluación de la expresión. Por tanto, las proposiciones como

**rapidez := rapidez + 1**

son perfectamente válidas: determinar el valor actual de la variable *rapidez*, sumarle uno a ese valor y después sustituir el valor existente de *rapidez* por el resultado. En pocas palabras, esta proposición dice “sumar uno a *rapidez*”. Este ejemplo hace resaltar el hecho de que ¡el operador de asignación *no* implica igualdad!

Supóngase que se tienen las siguientes declaraciones de variables en un programa en Pascal:

**VAR**

```
núm, suma : integer;
tiempo      : real;
calif       : char;
indicador   : Boolean;
```

Si se desea asignar valores a estas variables, debe tenerse cuidado de que los valores sean del tipo apropiado. Por ejemplo, no tiene sentido intentar asignar un valor booleano a una variable de carácter, y por tanto, no se permite en un programa en Pascal. Todas estas asignaciones son correctas:

```
calif := 'A'
indicador := false
tiempo := 20.15
núm := 0
suma := núm + 1
suma := suma + 1
```

Para comparar, he aquí unas cuantas proposiciones de asignación incorrectas con las mismas variables:

<i>proposición incorrecta</i>	<i>razón</i>
calif := B	Faltan apóstrofes en la constante de carácter 'B'.
núm := 1.0	Aquí núm es entera; 1.0 es real.
núm := tiempo	Aquí núm es entera; tiempo es real.
2 := núm	La variable debe estar a la izquierda.
sum + núm := time	La variable debe estar a la izquierda.

Existe una excepción a la regla que requiere que el tipo de datos de la variable de la izquierda y del valor de la expresión de la derecha sean idénticos. Se permite escribir una proposición de asignación que ordene a la computadora almacenar un valor entero en una variable real. Por ejemplo, la proposición

**tiempo := 60**

es legal, aun cuando se declaró *tiempo* (véase arriba) como variable real y 60 es una constante entera. Esto se debe a que todas las constantes enteras tienen una constante real que es equivalente. La computadora ejecuta la proposición que se muestra arriba como si se hubiera escrito la proposición

**tiempo := 60.0**

Esta conversión de entero a real es automática y se puede utilizar aunque el lado derecho del operador de asignación incluya una variable entera. Por ejemplo, la proposición

**tiempo := núm**

es legal y ordena a la computadora obtener el valor de *núm*, convertirlo en real y cambiar el valor actual de *tiempo* por este valor. Esta mezcla de tipos de datos puede ocurrir también en expresiones aritméticas, y se verán las consecuencias en la siguiente sección.

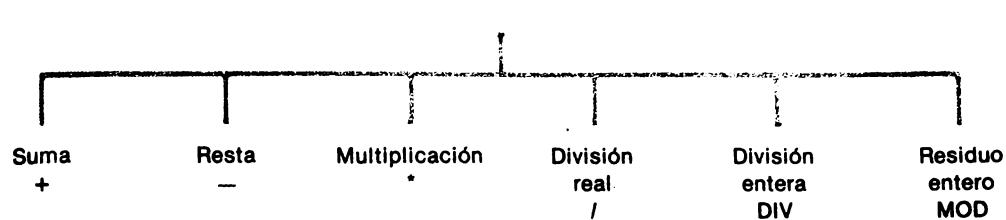
## Expresiones aritméticas

Considérese la siguiente expresión aritmética simple:

**5 + 2**

El símbolo + representa la adición y es un *operador aritmético*. Los valores 5 y 2 se llaman *operando*s. El valor de la expresión 5 + 2 se denomina *resultado* de la expresión.

Los cálculos con datos del tipo real o entero utilizan los siguientes operadores aritméticos (véase la Fig. 2-14):



**Figura 2-14** Operadores aritméticos del Pascal.

símbolo	significado	tipos de operandos	tipo del resultado
+	Sumar	Entero o real	Entero o real
-	Resta	Entero o real	Entero o real
*	Multiplicación	Entero o real	Entero o real
/	División	Real <sup>1</sup>	Real
DIV	División	Entero	Entero
MOD	Módulo (residuo)	Entero	Entero

<sup>1</sup>Se pueden emplear operadores enteros con el operador de división /, pero se convertirán de enteros en reales antes de la división.

Las expresiones que incluyen variables, constantes sin signo,<sup>2</sup> y cualquiera de estos operadores se llaman **expresiones aritméticas**. Los operadores se colocan entre **operandos** que se deben evaluar antes de que se pueda determinar el valor que resulta de la aplicación del operador. Por ejemplo, en la expresión

$a + b$

se debe determinar el valor de ambas variables (operandos) antes de que se pueda efectuar la adición. En la mayor parte de los casos, el Pascal espera (y requiere) que los tipos de las variables de una expresión sean los mismos. Sin embargo, aquí también es válida la excepción antes mencionada que permite el uso de datos de tipo entero en situaciones en las que se esperan datos reales. Para ilustrar esto, supóngase que se tienen las siguientes declaraciones:

**VAR**

```
núm1, núm2 : integer;
impuesto, costo : real;
```

<sup>2</sup>Si se permitieran constantes con signo, entonces sería válida una expresión como  $a - -4$ . Cabe hacer notar que si se define  $b$  (en una declaración de constantes) como constante cuyo valor es  $-4$ , entonces sí se permite  $a - b$ .

Entonces las siguientes expresiones son legales, aun cuando requieren la conversión de los valores de las variables enteras a valores reales;

costo + num1  
3.5 \* num2  
num1 / num2

El resultado de cada una de estas expresiones es un número real. Recuérdese que los valores enteros se pueden convertir en valores reales, pero los valores reales no se pueden convertir en enteros.

He aquí otras expresiones aritméticas válidas:

2 * num1 + num2	num2 MOD 5 - num1
impuesto / costo	num1 DIV num2

Recuérdese que la división entre cero es un error que se detecta cuando se ejecuta un programa que la incluye.

En Pascal existen dos operadores de división: / y DIV. La diferencia entre ellos es que el operador de división real (/) se puede usar con operandos enteros o reales y tiene un resultado real (incluso con operandos enteros). El operador de división entera (DIV) siempre produce un resultado entero; cualquier parte fraccionaria simplemente se ignora. Se puede utilizar el operador MOD para obtener el residuo de la división entera. También requiere operandos enteros y su resultado es entero. Por ejemplo, cuando se divide el entero 15 entre el entero 6, el cociente es 2 y el residuo es 3. La figura 2-15 muestra el resultado del algoritmo de división para obtener el cociente y el residuo.

Los siguientes ejemplos muestran los resultados que se obtienen cuando se utilizan los dos operadores de división y el operador MOD.

expresión	resultado	expresión	resultado
10.5 / 3.0	3.5	2.0 / 4.0	0.5
1 / 4	0.25	4 / 1	4.0
10 DIV 3	3	10 MOD 3	1
18 DIV 2	9	18 MOD 2	0
20 DIV 20	1	20 / 20	1.0
6 DIV 8	0	6 / 8	0.75

Ejemplos adicionales del operador MOD ilustrarán con detalle sus propiedades. MOD siempre tiene como resultado un entero positivo o cero, y el segundo operando no debe ser negativo o cero. En la mayor parte de las aplicaciones del

**Figura 2-15** Operadores DIV y MOD.

$$\begin{array}{r} 2 \xleftarrow{\quad} \text{Cociente } 15 \text{ DIV } 6 \\ 6) \overline{15} \\ \underline{12} \\ 3 \xleftarrow{\quad} \text{Residuo } 15 \text{ MOD } 6 \end{array}$$

operador MOD, ambos operandos son positivos, y se obtiene simplemente el residuo entero de la división. Sólo en el caso en que el primer operando sea negativo el resultado no será el residuo. En este caso el resultado de la expresión  $i \text{ MOD } j$  es  $i - (k * j)$ , donde  $k$  es un entero tal que la relación

$$0 \leq i - (k * j) < j$$

se cumple. He aquí algunos ejemplos:

expresión	resultado	razón
$-20 \text{ MOD } 100$	80	$80 = -20 - (-1 * 100)$
$20 \text{ MOD } 100$	20	residuo normal
$-20 \text{ MOD } -100$	error	segundo operando negativo

Desafortunadamente, algunas implantaciones de Pascal no dan el resultado correcto cuando el primer operando es negativo; el lector debe verificar que el Pascal que utiliza opere correctamente antes de suponer que los resultados del operador MOD son correctos.

Nótese que las siguientes expresiones son todas incorrectas por las razones indicadas:

expresión	razón
$25.0 \text{ DIV } 5$	Ambos operandos de DIV deben ser enteros.
$15 \text{ MOD } -5$	El segundo operando de MOD debe ser positivo.
$25.0 \text{ MOD } 5$	Ambos operandos de MOD deben ser enteros.
$17 / 0$	El segundo operando de / debe ser diferente de cero.
$17 \text{ DIV } 0$	El segundo operando de DIV debe ser diferente de cero.
$21 / -7$	Sólo se permiten constantes sin signo.

Es posible que el lector haya notado que no existe operador en Pascal estandar para la exponenciación (es decir, para elevar un número a otra potencia). Pascal sí cuenta con lo necesario para llevar a cabo la exponenciación en forma de funciones incluidas o estándar. Las funciones incluidas se estudian más adelante en este capítulo y la exponenciación es tema de un ejercicio al final del capítulo. Muchas versiones extendidas del Pascal sí incluyen un operador de exponenciación.

### Proposiciones de asignación y expresiones aritméticas

Como ya se vio, no se puede asignar un valor real (por ejemplo 6.5) a una variable entera. Si se pudieran realizar tales asignaciones, sería necesario considerar lo que sucedería a la parte fraccionaria del valor real. Hay dos opciones que se presentan inmediatamente: se podría simplemente hacer caso omiso de la parte fraccionaria y utilizar tan sólo la parte entera del número (lo que se denomina **truncar**) o se podría *redondear* el valor real hacia arriba o hacia abajo al valor entero más cercano. En realidad, ninguna de estas dos cosas se permite en Pascal, pero otros

lenguajes de programación sí las admiten. El truncado y redondeo se pueden realizar en Pascal, y más tarde se retomarán estos conceptos.

Como se mencionó, se pueden asignar valores enteros (como 12) a variables reales, y automáticamente se realiza la conversión del valor entero a su contraparte real (12.0). Las siguientes declaraciones y proposiciones de asignación ilustran ejemplos válidos, así como no válidos, de esta asignación de modo mixto.

## VAR

```
núm1, núm2, núm3 : integer;
total : real;
calif : char;
```

<i>proposición de asignación</i>	<i>comentarios</i>
total := núm1 + núm2 + núm3	El resultado entero se convierte en real
total := núm1 + 3.5	núm1 se convierte en real
total := núm1 DIV núm2	El resultado entero se convierte en real
total := núm1 / núm2	núm1 y núm2 se convierten en reales
núm := total + núm2	No válida, la expresión es real
núm3 := núm1 / núm2	No válida, la expresión es real
núm1 := total DIV 2	No válida, DIV requiere operandos enteros
calif := '9' + '7'	No válida, no se pueden sumar caracteres
calif := núm1	No válida, la expresión es entera

Conviene recordar los siguientes puntos:

- Toda expresión (o variable, o constante) tiene un tipo.
- Las expresiones que contienen valores tanto enteros como reales se evalúan con aritmética real, y para ello se convierten primero los operadores enteros en reales; el resultado de tales expresiones es un valor real.
- Las proposiciones de asignación requieren que la variable a la izquierda del operador de asignación tenga exactamente el mismo tipo que el valor que resulta de la expresión a la derecha del operador de asignación (pero si se pueden asignar expresiones enteras a variables reales).

## Orden de las operaciones aritméticas

Las expresiones aritméticas en las que se usa más de un operador (como  $2 * 3 + 5$ ) pueden evaluarse de varias maneras diferentes, según el operador que la computadora aplique primero. Dado que no conviene introducir este elemento de incertidumbre en los programas en Pascal, se utiliza un *orden de evaluación* definido en la evaluación de expresiones en Pascal. Por ejemplo, la expresión  $2 * 3 + 5$  se evaluaría al ejecutar primero la multiplicación  $2 * 3$  y sumar después 5 al resultado:

$$2 * 3 + 5 = 6 + 5 = 11$$

El orden en que se aplican los diversos operadores aritméticos de Pascal a sus operandos se especifica en forma única en la *jerarquía* de los operadores disponibles. Esta jerarquía es muy similar a la que se emplea en el álgebra y aritmética normales.

En Pascal, los diversos operadores de multiplicación y división (\*, /, DIV y MOD) se ejecutan antes de los operadores de suma y resta (+ y -). Por tanto, se dice que los operadores de multiplicación y división tienen *prioridad* sobre los operadores de suma y resta.

En una expresión que incluye más de un operador con la misma prioridad (como 6 DIV 3 \* 4), los operadores se deben aplicar en orden de izquierda a derecha. Por tanto, esta expresión daría el resultado entero 8 (6 DIV 3 es igual a 2 y 2 \* 4 es igual a 8), no cero (3 \* 4 es igual a 12 y 6 DIV 12 es igual a cero).

En ocasiones, el orden normal en que se aplican los operadores no es apropiado para una situación particular. Pascal permite el uso de paréntesis para forzar la evaluación de ciertas subexpresiones como una unidad. Como ejemplo, considérese la expresión 6 DIV (3 \* 4). Aquí se obliga a la computadora a efectuar la evaluación de 3 \* 4 antes de realizar la división.

Por supuesto, es posible “anidar” paréntesis (agrupar unos dentro de otros) para indicar el orden de evaluación en forma todavía más explícita. La expresión del ejemplo anterior se podría haber escrito 6 DIV (3 \* (2 + 2)), y produciría exactamente el mismo resultado. Cabe destacar sin embargo, que en Pascal no está definido explicitamente el orden de evaluación de los *operandos* de los operadores; en la expresión (3 + 4) DIV (2 + 1) podría evaluarse primero cualquiera de las dos partes en paréntesis. En este caso no importa si se evalúa primero 3 + 4 o 2 + 1, ya que es preciso efectuar ambas operaciones antes de la división. En otros casos, como se verá más adelante, el orden de evaluación puede ser importante.

En resumen, se ha visto que la evaluación de expresiones aritméticas se lleva a cabo según el siguiente algoritmo:

#### *Orden de aplicación de operadores*

- PASO 1    Evaluar cualquier expresión encerrada entre paréntesis.*
- PASO 2    Realizar las operaciones indicadas por los diversos operadores de multiplicación y división (\*, /, DIV y MOD). Si aparecen en secuencia dos o más de estos operadores (como en  $a * b / c$ ), los operadores se aplican de izquierda a derecha.*
- PASO 3    Realizar las operaciones indicadas por los operadores de suma y resta (+ y -). Si dos o más operadores aparecen en secuencia (como en  $a - b + c$ ) los operadores se aplican de izquierda a derecha.*

Conviene dar algunos ejemplos de evaluación de operaciones para ilustrar estas reglas. En estos ejemplos se utilizan únicamente constantes enteras para hacer más claros los pasos de evaluación. Naturalmente, pueden aparecer constantes reales y variables enteras y reales como operandos de los operadores aritméticos. Recuérdese, empero, que si se utilizan un operando entero y uno real con cual-

quier operador que permita operandos reales (específicamente  $+$ ,  $-$ ,  $*$  y  $/$ ), el operando entero se convertirá automáticamente en real y el resultado será real.

**Ejemplo 2.1**

$$5 \text{ MOD } 2 + 14 \text{ DIV } 3 - 6$$

$$\begin{array}{ll} 1 + 4 - 6 & (\text{MOD y DIV antes de } + \text{ y } -) \\ 5 - 6 & (\text{regla de izquierda a derecha}) \\ -1 & \end{array}$$

**Ejemplo 2.2**

$$5 + 2 * (3 + 7)$$

$$5 + 2 * 10 \quad (\text{primero la parte en paréntesis})$$

$$5 + 20 \quad (* \text{ antes de } +)$$

$$25$$

**Ejemplo 2.3**

$$3 * (4 \text{ MOD } (6 \text{ DIV } 2)) + 5$$

$$3 * (4 \text{ MOD } 3) + 5 \quad (\text{parte del paréntesis interior})$$

$$3 * 1 + 5 \quad (\text{parte del paréntesis exterior})$$

$$3 + 5 \quad (* \text{ antes de } +)$$

$$8$$

Conviene que el lector estudie las siguientes expresiones para comprobar su comprensión de la evaluación de expresiones aritméticas.

expresión	valor
12 DIV 5 * 3	6 (entero)
6 * 5 / 10 * 2 + 10	16.0 (real)
(6 * 5) / (10 * 2) + 10	11.5 (real)
(6 * 5) / (10 * 2 + 10)	1.0 (real)
(6 * 5) / (10 * (2 + 10))	0.25 (real)

**Traducción de expresiones algebraicas a Pascal**

Muchos problemas en la ciencia y la industria requieren el uso de expresiones y fórmulas algebraicas, por ejemplo, la fórmula cuadrática. Si se desea emplear Pascal para ayudar a resolver este tipo de problemas, es menester traducir las expresiones algebraicas a la forma de expresiones que reconoce Pascal.

El primer paso para traducir cualquier expresión algebraica a su equivalente en Pascal es verificar que cada operación (suma, resta, multiplicación y división) esté indicada explícitamente. Por ejemplo, la expresión algebraica  $5x + y$  es perfectamente aceptable en álgebra, ya que los dos términos  $5$  y  $x$  están realmente separados por un operador de multiplicación implícito. En Pascal, empero, no se permiten operadores implícitos, por lo que es necesario escribir de nuevo la expresión así:  $5 * x + y$ .

El segundo paso en la traducción de formato algebraico a Pascal se refiere a las expresiones algebraicas que requieren más de un renglón. Las más comunes son las que incluyen divisiones y exponenciaciones, como en la expresión

$$\frac{a + b^2}{a - b^2}$$

Las expresiones en Pascal se deben escribir totalmente en forma horizontal, por lo que la división y exponenciación en esta expresión se deben indicar como sigue:

$$(a + b * b) / (a - b * b)$$

Nótese que esto presupone que se desea realizar la división real, no la división entera. A continuación se presentan algunos ejemplos más de expresiones algebraicas y sus equivalentes en Pascal.

<i>expresión algebraica</i>	<i>expresión en Pascal</i>
5 (número + total)	5 * (número + total)
$a^2 + b^2$	$a * a + b * b$
$\frac{x + y}{u + w/a}$	$(x + y) / (u + w / a)$
$\frac{x}{y} (z + w)$	$x / y * (z + w)$

Las expresiones que se muestran en seguida no son aceptables en Pascal por las razones indicadas.

<i>expresión incorrecta</i>	<i>razón</i>
número * + total	Debe aparecer un operando entre cada par de operadores.
$x(y + z)$	Se debe indicar explícitamente la multiplicación.
$((a + b) * 4 - 3) / 2$	Los paréntesis deben estar equilibrados.
$\{(a + b) - 2\} / 4 - 1\} + 2$	Sólo se pueden emplear paréntesis para agrupar subexpresiones.

## Funciones estándar (incluidas)

Además de los diversos operadores aritméticos ya estudiados, Pascal cuenta con algunas funciones estándar o incluidas. La mayor parte de estas funciones tienen equivalentes exactos en álgebra. Se puede utilizar, por ejemplo, la función *sqr*

para obtener la raíz cuadrada de cualquier número no negativo. Existen otras funciones para determinar el seno de un ángulo y el valor absoluto de un número arbitrario. En el apéndice D se puede encontrar una lista completa de las funciones estándar con que cuenta Pascal. En esta sección se analizarán únicamente unas cuantas de éstas; las demás se irán estudiando conforme se necesiten.

La funciones estándar son de hecho funciones que ya se escribieron (basadas en el algoritmo apropiado), compilaron y probaron exhaustivamente para que el usuario no tenga que hacerlo. No es preciso ver las proposiciones del programa que realiza la función para poder usarla. Puede hacerse una analogía con el teléfono: no es indispensable comprender exactamente cómo realiza su función el teléfono o qué componentes contiene para poder hacer una llamada a través de él.

Lo que sí es necesario es poder solicitar en forma apropiada el uso de la función. En Pascal se dice que se *invoca* (o “llama”) una función cuando se solicita su uso. Supóngase, por ejemplo, que se desea calcular la raíz cuadrada de 100.6 y almacenar el resultado en una variable real llamada *raíz*. Esto puede hacerse con la siguiente proposición de asignación:

**raíz := sqrt (100.6)**

La expresión en esta proposición de asignación consiste totalmente en una invocación de función. Si se examina con cuidado la invocación de la función de raíz cuadrada, se pueden distinguir los siguientes detalles:

- Se escribe primero el nombre de la función, *sqrt*.
- En seguida viene una expresión encerrada entre paréntesis. Esta expresión se denomina *argumento* de la función.

El resultado de la función (en este caso 10.03) sustituye efectivamente a la invocación de la función. Es como si se hubiera escrito

**raíz := 10.03**

en vez de la proposición original. Por supuesto, la proposición en sí no ha cambiado realmente; la evaluación de expresiones que incluyen invocaciones de funciones se lleva a cabo como si se escribiera el resultado de la función en vez de su invocación. Otro ejemplo puede aclarar este punto. Supóngase que *x* y *y* son variables reales. La proposición de asignación

**x := 2 \* sqrt (y)**

asignará a *x* un valor igual a dos veces la raíz cuadrada de *y*. Por ejemplo, si la variable *y* valiera 169.0, entonces la función *sqrt* daría como resultado 13.0 (ya que  $13.0 * 13.0$  es igual a 169.0) y la proposición de asignación se manejaría como si se hubiera escrito

**x := 2 \* 13.0**

que asigna entonces a  $x$  el valor 26.0. Si el valor de  $y$  (el argumento) hubiera sido diferente de 169.0, la proposición de asignación hubiera almacenado un valor diferente en  $x$ .

El argumento de una función puede ser cualquier expresión válida, siempre que el tipo de la expresión del argumento (real, entero, etc.) corresponda al tipo que espera la función. En el caso de la función de raíz cuadrada, el argumento puede ser real o entero, pero el resultado siempre es real. Si se emplea una expresión del tipo incorrecto, el compilador de Pascal exhibirá un mensaje de error apropiado y no permitirá la ejecución del programa.

Dos funciones más que incluye el Pascal son *round* y *trunc*. Estas funciones aceptan únicamente argumentos reales, y el resultado es entero. (Se les llama muchas veces funciones de *transferencia*, ya que transfieren, o convierten, un valor de un tipo a otro.) Como podría esperar el lector, la función *round* (redondear) da como resultado el entero más cercano a su argumento. Por ejemplo:

expresión	resultado
<i>round</i> (5.5)	6
<i>round</i> (-3.5)	-4
<i>round</i> (8.7)	9
<i>round</i> (2.1)	2
<i>round</i> (-3.1)	-3

Nótese la diferencia en el redondeo de números positivos y negativos.

La función *trunc* da como resultado el entero que queda al omitir, o truncar, la parte fraccionaria del número real. Por ejemplo:

expresión	resultado
<i>trunc</i> (5.6)	5
<i>trunc</i> (2.1)	2
<i>trunc</i> (-2.8)	-2

La función *sqr* da como resultado un valor igual al cuadrado de su argumento con el mismo tipo que el argumento. Así, *sqr* ( $a$ ) es tan sólo un notación breve de “ $a * a$ ”, donde  $a$  es el argumento de la función. Por ejemplo:

expresión	resultado
<i>sqr</i> (3)	9 (entero)
<i>sqr</i> (3.0)	9.0 (real)

La última función estándar que se estudiará en esta sección es *abs*, que produce el valor absoluto (magnitud) de una expresión. Al igual que *sqr*, el tipo de su resultado es el mismo que el tipo de su argumento. Los siguientes ejemplos ilustran sus propiedades:

expresión	resultado
abs (8)	8 (entero)
abs (-10)	10 (entero)
abs (5.4)	5.4 (real)

Puesto que las invocaciones de funciones aparecen en expresiones, es preciso obtener los valores que representan antes de que se puedan aplicar operadores a esos valores. En la expresión

`a * sqrt (b) / abs (c)`

es obvio que no puede efectuarse la multiplicación hasta que no se haya obtenido el valor de la variable *a* y la raíz cuadrada de la variable *b*. Una vez ejecutada la multiplicación, no se puede efectuar la división mientras no se haya calculado el valor absoluto de la variable *c*. Recuérdese ahora lo que se apuntó acerca del orden de evaluación de los *operандos*. En la expresión

`sqrt (a) / sqrt (b)`

se podría realizar primero cualquiera de las dos raíces cuadradas. Normalmente esta propiedad (es decir, la falta de un orden específico de evaluación de operandos) no causa problemas, pero se volverá más tarde a este tema cuando se vuelva importante.

Éstas son, en resumen, las propiedades de las funciones estándar:

- Toda función estándar ejecuta un algoritmo ya probado para calcular valores útiles.
- El tipo del argumento debe concordar con el que espera la función.
- El tipo del resultado que produce una función depende de la función de que se trate y, a veces, del tipo del argumento.
- Las expresiones que se usan como argumentos se deben evaluar (conforme las reglas usuales) antes de que se invoque a la función.
- Los nombres de las funciones estándar ya están declarados (son identificadores estándar). Si se declaran explícitamente, los identificadores se referirán entonces a los objetos explícitamente declarados.
- El orden en que se evalúan las expresiones determina el orden en que se realizan las invocaciones de funciones.

He aquí varios ejemplos adicionales que utilizan funciones estándar.

#### Ejemplo 2.4

Evaluar  $12 + \sqrt{\sqrt{12} - 4 * 11}$ .

$$12 + \sqrt{\sqrt{12} - 44}$$

$$12 + \sqrt{144 - 44}$$

$$12 + \sqrt{100}$$

$$12 + 10$$

$$22$$

**Ejemplo 2.5**

Una solución de la ecuación cuadrática

$$ax^2 + bx + c = 0$$

está dada por la expresión algebraica

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}.$$

Esta expresión se escribe en Pascal así:

$$(-b + \text{sqrt}(b^2 - 4 * a * c)) / (2 * a)$$

Tómese nota de que todos los paréntesis en esta expresión son necesarios para representar correctamente la forma algebraica.

## EJERCICIOS DE LA SECCIÓN 2.4

- 1 Supóngase que se tienen las siguientes declaraciones de variables.

VAR

temp, valor : entero;  
núm, suma : real;

Determíñese cuáles de las siguientes proposiciones de asignación son válidas

- a)  $\text{núm} := \text{temp} + \text{valor}$     b)  $\text{valor} := \text{núm} + \text{temp}$   
c)  $\text{núm} := \text{suma}$                     d)  $\text{valor} := \text{temp} + 3$   
e)  $\text{valor} := \text{núm} + 3$               f)  $\text{núm} := \text{núm} * \text{suma}$   
g)  $\text{temp} := \text{núm} * \text{suma}$

- 2 Evalúense las siguientes expresiones.

- a)  $6 \text{ DIV } 2 - 6 \text{ MOD } 5$   
b)  $14 \text{ MOD } 2 * 6 + 3$   
c)  $3 + 14 \text{ MOD } (2 * 3)$   
d)  $5 \text{ MOD } 8 + 8 \text{ MOD } 5$   
e)  $7 \text{ MOD } 2 + 13 \text{ DIV } 3 - 2$   
f)  $6 - 2 * (1 + 4) + 5$

- 3 Supóngase que se tienen las siguientes declaraciones de variables:

VAR

ventas, total, pérdida : entero;  
calif : real;

Determine cuáles de las siguientes proposiciones de asignación son válidas.

- a) calif := ventas + total + pérdida
- b) ventas := ventas + 5.0
- c) calif := ventas DIV total
- d) calif := ventas / total
- e) ventas := calif / total
- f) total := calif DIV pérdida

4 Evalúense las siguientes expresiones.

- (a) sqrt (16)      (e) trunc (-13.8)
- (b) round (10.7)      (f) sqr (5)
- (c) round (-3.8)      (g) abs (-12)
- (d) trunc (10.1)      (h) trunc (8.6) – round (8.6)

5 Supóngase que *acosto*, *bcosto*, *ccosto* y *dcosto* son variables reales con los siguientes valores:

```
acosto := 4.0;
bcosto := 1.0;
ccosto := -2.0;
dcosto := 5.5;
```

Evalúense las siguientes expresiones.

- a) sqrt (acosto /bcosto – ccosto + dcosto – 2.5)
- b) trunc (dcosto) \* abs (ccosto \* (bcosto / acosto))

6 Escríbase una expresión en Pascal para cada una de las siguientes expresiones matemáticas.

$$(a) \frac{x + y}{\frac{y}{z} + 3}$$

$$(b) \frac{-b - \sqrt{b^2 - 4ac}}{2a} \quad (c) (x^2 + y^2)^2$$

$$(d) \frac{x + y - z}{2x + y^2 - xyz}$$

7 Escríbase una expresión matemática equivalente a cada una de las siguientes expresiones en Pascal.

- (a) a \* b \* c / d / e
- (b) x + y / z \* (a + b)
- (c) b \* b – 4 \* a \* c
- (d) a – b \* c / d + e \* f

8 Insértense paréntesis en cada una de las siguientes expresiones en Pascal para indicar el orden en que se aplicarán los operadores.

- (a)  $2 * b - a + b * d / e$
- (b)  $f - e * d / c + b * a$
- (c)  $a * a - b * c / d + \text{sqrt}(e + 2 * f)$
- (d)  $d * (a + b) - \text{trunc}(e * f / g) + b / (a - b * c)$
- (e)  $6 \text{ DIV } 2 * 5 \text{ MOD } 3 - 2 * 3$
- (f)  $2 * 3 \text{ MOD } (8 \text{ DIV } 3 + 1) + \text{round}(6.5)$

**9** Determíñese si las siguientes son proposiciones de asignación válidas en Pascal. Supóngase que *núm*, *suma* y *total* son variables enteras, *valor* es una variable real y *car1* y *car2* son variables de carácter.

- a)  $\text{núm} := \text{núm} + \text{núm}$
- b)  $\text{núm DIV suma} / \text{total}$
- c)  $\text{suma} := \text{núm} / \text{total}$
- d)  $\text{valor} := \text{total}$
- e)  $\text{valor} := \text{total} * \text{núm} + \text{sum}$
- f)  $\text{valor} := \text{total} + \text{car1}$
- g)  $\text{car2} := \text{car1}$
- h)  $\text{car2} := \text{car1} + 1$
- i)  $\text{car2} := \text{'car1'}$

**10** Evalúense las siguientes expresiones en Pascal.

- (a)  $5 + \text{sqr}(3) - 4 + \text{trunc}(3.6 - 2.1)$
- (b)  $3 * \text{sqrt}(8 \text{ MOD } 6 * 10 \text{ DIV } 5)$
- (c)  $6 + 9 * 8 \text{ DIV } 2 * \text{round}(1.362) - 2 * 3$
- (d)  $\text{trunc}(12 / 5 * \text{sqrt}(4 + 4 * 3) / 4)$

## SECCIÓN 2.5 TÉCNICAS DE PRUEBA Y DEPURACIÓN

Cuando se realizan operaciones aritméticas en Pascal, el programador debe tener cuidado con ciertos errores que se pueden presentar. Por ejemplo, supóngase que la escala de números reales positivos permitida en un sistema de cómputo determinado es de  $10^{-38}$  a  $10^{38}$ . Si un cálculo da por resultado un valor mayor que  $10^{38}$ , lo más probable es que el sistema de cómputo exhiba un mensaje que indique un *desborde* (*overflow*) y detenga inmediatamente la ejecución del programa. En forma similar se presentará una *insuficiencia* (*underflow*) si el valor calculado es demasiado pequeño (es decir, mayor que cero pero con una magnitud menor de  $10^{-38}$ ). La mayor parte de los sistemas no manejan la insuficiencia de la misma manera que tratan el desborde, sino que producen un resultado de exactamente cero.

Otro problema que se presenta en los cálculos con números reales es la pérdida de *exactitud*. El número de dígitos que se emplea para representar un número real en la notación científica depende de la máquina de que se trate. La *precisión* de la máquina se define como el número máximo de dígitos que puede emplear la computadora para representar un número. Puesto que los valores reales son números

aproximados limitados por la precisión de la computadora, puede haber pérdida de exactitud, lo que podría causar errores significativos en los cálculos.

Considérese el siguiente caso. En teoría, la expresión  $(0.00000234 - 2.0) + 2.0$  debería dar el valor 0.00000234, ya que  $-2.0 + 2.0$  es exactamente cero; pero supóngase que esta expresión se evalúa en un sistema de cómputo con una precisión de cinco dígitos decimales. Puesto que los paréntesis indican que se debe efectuar primero la resta, la computadora debe evaluar primero  $0.00000234 - 2.0$ . ( Nótese que 0.00000234 tiene únicamente tres dígitos de precisión.) Con una exactitud de cinco decimales, el resultado es  $-2.0000$ . Ahora, cuando se hace la adición, la computadora produce un resultado de exactamente cero, no 0.00000234. La razón de esta pérdida de exactitud es la combinación de números muy pequeños con otros relativamente grandes.

He aquí varios aspectos importantes que conviene tomar en cuenta al probar y depurar cálculos aritméticos:

- Asegúrese de que todas las variables se declararon.
- Asegúrese de que los tipos de datos concuerdan en las proposiciones de entrada y salida y en las proposiciones de asignación.
- Pruebense a mano los cálculos con algunos ejemplos sencillos.
- Verifíquese que la forma que se emplea para los datos de entrada y las proposiciones *read* y *readln* sean consistentes (véase el siguiente capítulo).
- Revísese el orden en que se evaluarán las expresiones aritméticas; si se tienen dudas, utilíicense paréntesis para garantizar el orden de evaluación.

### **Introducción, modificación y ejecución de programas en Pascal**

Antes de intentar escribir sus propios programas en Pascal, casi siempre es conveniente que el lector introduzca o modifique un programa ya existente. Esto le ayudará a entender los detalles de edición, compilación y ejecución de un programa en su propio sistema. En los ejercicios del capítulo se encontrarán problemas para resolución en computadora que pedirán al lector que modifique un programa en Pascal ya existente y más tarde realice todos los pasos necesarios para ejecutarlo. En el siguiente capítulo se analizarán los detalles de entrada, salida y resolución de problemas en Pascal. Después de completar ese capítulo, el lector será capaz de escribir sus propios programas completos en Pascal.

Como ayuda para probar y depurar programas se da a continuación una lista de recordatorios importantes de Pascal.

## **RECORDATORIOS DE PASCAL**

### **Comentarios**

- Deben estar delimitados por las parejas (\* y \*), o por llaves {y}.

### **Símbolo de punto y coma**

- Separa proposiciones. No es necesario un punto y coma antes de END.

```

PROGRAM ejemplo (input, output);
VAR a, b, c : integer;
BEGIN
    proposición;
    proposición;
    .
    .
    .
    proposición (* no hay punto y coma *)
END.
```

***Programas en Pascal***

- Deben terminar con un punto.
- Deben tener las proposiciones ejecutables encerradas entre las palabras BEGIN y END.
- Es conveniente que incluyan documentación en forma de comentarios y variables y constantes que se documenten a sí mismas.

***Constantes***

- Se declaran antes de la declaración de variables.
- No se pueden cambiar durante la ejecución de un programa.

***Variables***

- Deben declararse en los programas en Pascal usando la palabra reservada VAR.
- Sólo se les pueden dar valores del tipo declarado.
- Deben tener un valor antes de que se puedan utilizar en una expresión.

***Números***

- No deben incluir espacios o comas.
- Si tienen punto decimal deben tener por lo menos un dígito antes y uno después del punto decimal:

0.05                  -2.0

- Los de tipo entero pueden mezclarse libremente con números del tipo real en expresiones aritméticas y se convertirán automáticamente en los valores reales equivalentes en esos casos.
- Los de tipo real se pueden convertir en enteros por medio de las funciones estándar *round* y *trunc*.

***Caracteres***

- Se encierran entre apóstrofos:

'A'                  'b'                  '8'

- apóstrofo se representa como dos apóstrofos consecutivos:

'''

***Proposición de asignación***

- Asigna el valor de una expresión a una variable.
- Requiere que la variable “de destino” y la expresión tengan el mismo tipo (con la excepción de que se pueden asignar expresiones enteras a variables reales).
- Ejemplos:

A := 2 \* B + 1

cuenta := cuenta + 1

***División***

- De reales mediante el operador /
- De enteros mediante el operador DIV
- El residuo (entero) se obtiene mediante el operador MOD

***Orden de evaluación***

- Primero se evalúan las expresiones dentro de paréntesis.
- Despues se realizan las multiplicaciones y divisiones, de izquierda a derecha.
- Despues se realizan las sumas y restas, de izquierda a derecha.

**SECCIÓN 2.6 REPASO DEL CAPÍTULO**

En este capítulo se analizaron detalles del lenguaje Pascal. Un programa en Pascal consta de dos componentes: un encabezado de programa y un bloque de programa. El encabezado de programa es una sola proposición que contiene el nombre del programa y una lista de las variables de archivo que se conectarán a archivos externos cuando se ejecute el programa. El bloque de programa consiste en las declaraciones (p. ej. de variables y constantes) y las proposiciones ejecutables. La estructura general de los programas en Pascal de este capítulo tiene la siguiente forma:

```

PROGRAM nombre (input, output);
CONST declaraciones de constantes;
VAR declaraciones de variables;
BEGIN
  proposición;
  proposición;
  ...
  proposición
END.

```

He aquí un resumen de los detalles de Pascal que se analizaron en este capítulo. Se recomienda al lector usar el resumen como referencia en el futuro.

**REFERENCIAS DE PASCAL****1 Identificadores, constantes y variables**

- 1.1 *Identificador*: palabra o secuencia de caracteres que constituye el nombre de un objeto de programa. Los identificadores son secuencias de letras y dígitos que deben comenzar con una letra:

impventa      H20

*Palabra reservada*: identificado: reservado para un objetivo especial en el programa en Pascal; esos identificadores no pueden volverse a definir.

BEGIN      END      VAR

*Identificador estándar*: identificador previamente declarado por el compilador que puede redefinirse (aunque no es recomendable):

input      output      read      integer

- 1.2 *Constante*: valor declarado que no cambia durante la ejecución del programa y que se especifica mediante la palabra reservada CONST:

CONST

tasaimp = 0.05;  
pi = 3.14159; .

- 1.3 *Variable*: objeto declarado que tiene un valor susceptible de ser cambiado durante la ejecución y que se especifica mediante la palabra reservada VAR:

VAR

distanzia, tiempo : real;  
velocidad : integer;

- 2 Tipos de datos simples: entero, real, booleano y de caracteres

- 2.1 *Entero (integer)*: números enteros positivos y negativos, incluso el cero:

-60      45      0      10

*Maxint*: el entero más grande que puede representar el compilador de Pascal específico que se usa.

- 2.2 *Real*: números con punto decimal y/o exponente:

0.5      -2.4      6.3e-10

- 2.3 *De carácter (char)*: caracteres como letras, dígitos y símbolos especiales:

'A'      'B'      '?'

- 2.4 *Booleano (Boolean)*: tipo de datos que tiene únicamente dos valores, *true* (verdadero) y *false* (falso).

- 3 Proposición de asignación:** asigna el valor de la expresión que está a la derecha del operador de asignación (`:=`) a la variable que se especifica en el lado izquierdo del operador (los componentes en ambos lados del operador deben ser del mismo tipo):

```
tiempo := 60;
calif := 'A';
núm := núm + 1;
```

#### 4 Expresiones aritméticas

##### 4.1 Operadores aritméticos:

<code>+</code>	Suma
<code>-</code>	Resta
<code>*</code>	Multiplicación
<code>/</code>	División (cociente real)
<code>DIV</code>	División (cociente entero)
<code>MOD</code>	División (residuo entero)

##### 4.2 Orden de los operadores:

- 1) Primero se evalúan las expresiones encerradas entre paréntesis.
- 2) Despues se aplican los operadores de multiplicación y/o división (`*, /, DIV, MOD`) y la evaluación se hace de izquierda a derecha cuando hay dos o más de estos operadores.
- 3) Los operadores de suma, y resta se aplican al último, de izquierda a derecha cuando hay dos o más de estos operadores.

##### 4.3 Funciones estándar (incluidas): funciones previamente definidas que puede utilizar un programa en Pascal:

<code>sqrt(x)</code>	Raíz cuadrada de $x$
<code>round(x)</code>	Redondea el argumento real $x$ a entero
<code>trunc(x)</code>	Trunca el argumento real $x$ a entero
<code>sqr(x)</code>	Eleva al cuadrado el argumento $x$
<code>abs(x)</code>	Valor absoluto del argumento $x$

En el apéndice D se proporciona una lista de otras funciones estándar.

## Avance del capítulo 3

En el siguiente capítulo se estudiarán los mecanismos que sirven para comunicar datos entre la computadora y el “mundo real”. Por ejemplo, con frecuencia se exhiben los resultados de los cálculos en un dispositivo de salida, y los valores de los datos para los cálculos se obtienen de teclados u otros dispositivos de entrada. Además de entrada y salida, se hablará de la resolución de problemas con Pascal.

argumento	identificador estándar
bloque de programa	jerarquía
booleano	lenguaje de tipos rígidos
cadena de caracteres	máxint
carácter, de ( <i>char</i> )	notación científica
comentario	notación de punto flotante
constante	operador aritmético
declaración	operando
delimitador	palabra reservada
diagrama de sintaxis	proposición de asignación
encabezado de programa	proposición ejecutable
entero ( <i>integer</i> )	real
exactitud	reglas de sintaxis
función de transferencia	tipo de datos
función estándar	truncar
identificador	variable

## EJERCICIOS DEL CAPÍTULO 2

### ★ EJERCICIOS ESENCIALES

- 1 Declárense variables con nombres descriptivos apropiados para representar el valor de lo siguiente:
  - a) El número de gatitos de una carnada
  - b) La clave en la que está escrita una composición musical determinada
  - c) Un número de vuelo de línea aérea con tres dígitos y una letra
  - d) El valor de pi dividido entre 2
  - e) La razón de dos valores numéricos arbitrarios
  - f) Si hoy es sábado o no
- 2 Considérese la proposición de asignación  $a := b$ , donde  $a$  es una variable real y  $b$  es una variable entera. Es evidente que se debe convertir el valor de  $b$  de entero a real antes de que se pueda almacenar en  $a$ , pero no se altera el contenido de la variable  $b$ . ¿Qué sugiere esto acerca de las formas en que se puede tener acceso a la memoria de las computadoras?
- 3 Determínese el resultado producido por el compilador de Pascal que utiliza el lector para las siguientes expresiones. ¿Cuáles resultados concuerdan con el resultado que se obtiene en Pascal estándar?
 

a) $5 \text{ MOD } 2$	b) $5 \text{ MOD } -2$
c) $-5 \text{ MOD } 2$	d) $-5 \text{ MOD } -2$
- 4 Supóngase que un número de tres dígitos tiene la forma  $abc$ . Por ejemplo, si el número fuera 730, entonces  $a$  representa al siete,  $b$  al tres y  $c$  al cero. Escríbanse expresiones en Pascal que produzcan los “números”  $cba$  y  $accb$ .

- 5 Determinese cuáles de las siguientes proposiciones son legales. Supóngase que se declararon apropiadamente todas las variables de manera que no existe incompatibilidad de tipos.

- a)  $v := a(*entera*) \text{DIV } b$
- b)  $v := a \text{DIV } b$
- c)  $v := a \text{ DIV } b$
- d)  $v(*a \text{ veces}*) := (*\text{aprox.}*) a \text{ DIV } b$
- e)  $s := ('*\text{Mensaje}*')$
- f)  $s := ('*\text{Mensaje}*')\text{'Número Dos'}$

- 6 Insértense paréntesis en las siguientes expresiones para que su significado quede más claro. No se debe cambiar el valor de las expresiones. Indíquese el tipo de cada expresión.

- a)  $4 + 6 \text{ MOD } b \text{ DIV } c$
- b)  $j \text{ DIV } k / 17.04e-5 / 2.0$
- c)  $1 + 2 \text{ DIV } 3 * 4 + 5 \text{ DIV } 6$
- d)  $2 / \text{sqrt}(17.03 - a \text{ DIV } 21 * j \text{ MOD } k) * 1.0001$

### ★ ★ EJERCICIOS IMPORTANTES

- 7 Algunos compiladores de Pascal (o sistemas de cómputo) hacen que se inicialicen las variables cuando se les asigna una localidad en la memoria. Para determinar si el Pascal que utiliza el lector es de este tipo, ejecútese varias veces el siguiente programa. Si se utiliza una microcomputadora, apáguese la computadora entre corridas.

```
PROGRAM verificar (output);
VAR
    i : integer;
    r : real;
BEGIN
    writeln (i, r)
END.
```

- 8 ¿Se le ocurren al lector otros tipos de datos que no sean enteros, reales, de caracteres y booleanos que pudieran ser útiles en un lenguaje de programación? ¿Se podrían crear valores para estos tipos de datos a partir de combinaciones de los tipos de datos estándar que ya existen en Pascal?
- 9 Experimente para determinar lo que sucede cuando se ejecutan las siguientes proposiciones de asignación:

```
muygrande := máxint + 1;
muychico := -máxint - 1;
```

¿Qué sugieren estos resultados acerca de la representación de enteros?

- 10 Desarróllese un sistema para obtener el exponente (potencia de 10) de un número real arbitrario. Por ejemplo, el exponente de 3.56E-10 es -10, y el exponente de 4.526E + 7 es 7. (Sugerencia: considérese el uso de la función *ln* para determinar el logaritmo base 10 de un número.)
- 11 La función estándar *ord* produce un entero que indica la posición relativa de cualquier constante ordinal (entera, de caracteres o booleana) dentro del conjunto de las constantes del mismo tipo. Determínese el valor ordinal de cada una de las siguientes constantes. (La función *ord* se examina en detalle en el Cap. 8.)
- a) true      b) 'A'      c) ';'      d) 0      e) -1      f) maxint  
 g) false      h) 'a'      i) "      j) ""      k) 1      l) -maxint
- 12 Matemáticamente,  $\ln a^b = b \cdot \ln a$ , y  $e^{\ln x} = x$ . Utilíicense estas propiedades y las funciones estándar de Pascal *ln* y *exp* para escribir una expresión en Pascal que produzca el valor de  $a^b$ . ¿Para qué valores de *a* y *b* deja de funcionar esta expresión?
- 13 Suponga el lector que decidió probar un programa al que se le ha quitado cierta parte. En vez de borrar esa porción del programa fuente, el lector opta por encerrarla entre (\* y \*) para convertirla en un comentario. De esta forma, la parte eliminada se puede restaurar simplemente si se quitan los delimitadores del comentario.
- a) ¿Por qué pensaría el lector que esto va a funcionar?  
 b) ¿Por qué no funcionará?
- 14 ¿Producen el mismo valor las expresiones

`sqr(sqrt(2.0))`      y      `sqrt(sqrt(2.0))`?

¿Está seguro de ello el lector? ¿Es siempre cero el valor de

`sqr(sqrt(2.0)) - sqrt(sqrt(2.0))`

¿Por qué?

### ★ ★ ★ EJERCICIOS ESTIMULANTES

- 15 Determínese el número real positivo más pequeño que se puede almacenar en una variable real. ¿Qué sucede cuando se divide entre dos ese número?

## PROBLEMAS DEL CAPÍTULO 2 PARA RESOLUCIÓN EN COMPUTADORA

Para adquirir cierta experiencia en la introducción, compilación y ejecución de programas en Pascal, intodúzcanse y ejecútense los siguientes programas. Se sugieren ciertos cambios adicionales para cada problema.

## ★ PROBLEMAS ESENCIALES

Introdúzcanse los siguientes programas. Trate el lector de anticipar cuál va a ser la salida. Después ejecútelo y compare lo que esperaba con la salida que se exhibe.

- 1 PROGRAM muestra1 (output);
 

```

CONST
    pi = 3.1415926535;
    r1 = 2.0;
    r2 = 5.0;
VAR
    área : real;
BEGIN
    área := pi * r1 * r1;
    writeln (r1, área); (*Exhibir valor de r1 y área*)
    área := pi * sqr (r2);
    writeln (r2, área); (*Exhibir valor de r2 y área*)
END.
```
- 2 PROGRAM muestra2 (input, output);
 

```

VAR
    núm : real;
    a, b : integer;
BEGIN
    readln (núm);    (* Obtener un valor para núm *)
    a := round (núm);
    b := trunc (núm + 0.5);
    writeln (núm, a, b)    (*Exhibir núm, a, y b *)
END.
```

Cuando el programa espere datos de entrada, intodúzcase un número real. Ejecútese el programa varias veces y úsense números tanto positivos como negativos con partes fraccionarias mayores, menores y exactamente iguales que 0.5.

## ★ ★PROBLEMAS IMPORTANTES

- 3 El siguiente programa calcula el promedio aritmético de cuatro números reales que se obtienen de la proposición de entrada de datos. Modifíquese para que en vez de ello calcule el promedio geométrico de los datos. El promedio geométrico de  $N$  datos,  $V_1, V_2, \dots, V_N$  se define así:

$$(V_1 * V_2 * \dots * V_N)^{1/N}.$$

```

PROGRAM promedio (input, output);
VAR
    v1, v2, v3, v4 : real;
    promedio : real;
```

```

BEGIN
    readln (v1, v2, v3, v4);      (* Obtener datos *)
    promedio := (v1 + v2 + v3 + v4) / 4.0;
    writeln ('Promedio =', promedio) (* Exhibir el promedio *)
END.

```

- 4 El siguiente programa convierte una temperatura real que se obtiene de los datos de entrada de Fahrenheit en Celsius. Modifíquese el programa para que convierta la temperatura de Celsius en Fahrenheit.

```

PROGRAM conversión (input, output);
CONST
    factor1 = 32.0;
    factor2 = 0.5555555555;      (* 5/9 *)
VAR
    fahrenheit,
    celsius : real
BEGIN
    readln (fahrenheit);
    celsius := factor2 * (fahrenheit - factor1);
    writeln (fahrenheit, celsius)
END.

```

### ★ ★ ★ PROBLEMAS ESTIMULANTES

- 5 El siguiente programa, que posiblemente no entienda por completo el lector en este momento, determina la longitud de los renglones más cortos y más largos en los datos de entrada, así como la longitud promedio de los renglones de entrada. Trátese de proporcionar entradas que produzcan la salida:

El renglón más corto tiene 1 caracteres  
 El renglón más largo tiene 10 caracteres  
 El renglón promedio tiene 5.00 caracteres

Después, trátese de crear datos que tengan por lo menos un renglón y que produzcan la salida:

El renglón más corto tiene 0 caracteres  
 El renglón más largo tiene 0 caracteres  
 El renglón promedio tiene 0.00 caracteres

PROGRAMA longrenglón (input, output);  
 VAR

más corto : integer; (\* longitud del renglón más corto \*)
 más largo : integer; (\* longitud del renglón más largo \*)
 total : integer; (\* número total de caracteres \*)
 número : integer; (\* número total de renglones \*)
 promedio : real; (\* promedio de caracteres por renglón \*)

```

largo : integer;      (* longitud de la línea actual *)
car : char;          (* carácter actual de entrada *)

```

```
BEGIN
```

```

máscorto := máxint;
más largo := 0;
número := 0;
total := 0;
WHILE NOT eof(input) DO
BEGIN
    largo := 0;
    WHILE NOT eoln(input) DO
    BEGIN
        read (car);
        largo := largo + 1
    END;
    readln;
    IF largo > más largo
    THEN más largo := largo;
    IF largo < máscorto
    THEN máscorto := largo;
    total := total + largo;
    número := número + 1
END;
IF número = 0
THEN BEGIN
    máscorto := 0;
    promedio := 0
END
ELSE promedio := total / número;
writeln ('El renglón más corto tiene ', máscorto:1,' caracteres');
writeln ('El renglón más largo tiene ', más largo:1,' caracteres');
writeln ('El renglón promedio tiene', promedio:4:2, 'caracteres')

```

```
END.
```

Ejemplo de entrada:

Columna 1

↓

Es cierto que son operaciones básicas.

Corto

Caracteres poco comunes

Estudia mucho

Ejemplo de salida:

El renglón más corto tiene 5 caracteres

El renglón más largo tiene 38 caracteres

El renglón promedio tiene 19.75 caracteres

# CAPÍTULO 3

CAPÍTULO 3  
ENTRADA, SALIDA Y  
RESOLUCIÓN DE  
PROBLEMAS EN PASCAL

ICA

Entrada y  
salida

Resolución  
de problemas  
con Pascal

Entrada y  
salida de  
archivos de  
texto  
(opcional)

12. Entrada y salida

## ENTRADA, SALIDA Y RESOLUCIÓN DE PROBLEMAS EN PASCAL

## OBJETIVOS

Después de completar este capítulo, el lector deberá ser capaz de:

- Reconocer y aplicar proposiciones de entrada/salida en Pascal, por ejemplo *read* y *write*
- Construir y documentar un programa sencillo en Pascal
- Resolver, probar y depurar un programa sencillo en Pascal
- En forma opcional, reconocer y aplicar la entrada y salida de archivos de texto

## PANORAMA GENERAL DEL CAPÍTULO

En este capítulo se analiza la tarea de introducir datos a la computadora y obtener resultados de ella. Sin la capacidad de introducir y sacar datos, la posibilidad de realizar cálculos rápida y exactamente carece de valor; la computadora debe poder comunicarse con el “mundo exterior”. Se examinarán algunos de los mecanismos de entrada y salida (E/S) con que cuenta el Pascal. Adicionalmente, se presentará un análisis detallado de la resolución de problemas mediante Pascal.

En la primera sección se estudia el problema de obtener datos numéricos y no numéricos de un dispositivo de entrada (por lo general un teclado) para que los utilice un programa en Pascal. También se examina el problema de las salidas, es decir, la exhibición de diferentes tipos de datos en una impresora o terminal. Se presentará en forma específica y detallada la forma de controlar el aspecto de las salidas (su formato). Una sección opcional analiza la entrada y salida de archivos almacenados en discos y cintas magnéticos.

El capítulo también incluye un análisis de las aplicaciones de Pascal a la resolución de problemas. Se presentan dos problemas y sus soluciones completas con base en las estrategias que se analizaron en el capítulo 1. Éstas incluyen análisis de problemas, resolución de problemas mediante el diseño descendente y resolución en computadora. Se incluye el programa en Pascal completo que resuelve cada uno de los problemas. La sección de técnicas de prueba y depuración se dedica a los errores comunes de entrada y salida. Después de completar este capítulo, el lector deberá ser capaz de construir y documentar un programa sencillo en Pascal.

## SECCIÓN 3.1 ENTRADA Y SALIDA

En esta sección se estudian los detalles de la lectura y escritura de datos en Pascal. Un programa puede operar sobre los datos que se proporcionan mediante un dispositivo de entrada. Los resultados se envían a un dispositivo de salida. Este análisis incluirá la manipulación de datos tanto numéricos como no numéricos.

Ya se vio que un método que se puede usar para cambiar el valor de una variable es la proposición de asignación. Otra forma de proporcionarle un valor a una variable es dárselo como dato de entrada. Pascal cuenta con dos proposiciones, *read* y *readln*, para transferir datos de un dispositivo de entrada a la memoria de la computadora<sup>1</sup>. Considérese la siguiente proposición:

**read (horas, min, seg)**

La proposición *read* va seguida de los identificadores de las variables, separados por comas y encerrados entre paréntesis. Supóngase que se declararon como enteras las variables *horas*, *min* y *seg*. La proposición *read* ordena a la computadora que obtenga tres enteros del dispositivo de entrada y almacene en orden los valores en las localidades de memoria que corresponden a las variables de la lista. Por ejemplo, si el dispositivo de entrada es el teclado de una terminal y el usuario teclea las constantes enteras.

3        25        30

entonces la computadora asignará el valor 3 a la variable *horas*, el valor 25 a la variable *min* y el valor 30 a la variable *seg*. Téngase en cuenta que los valores están separados por espacios. También se acepta un carácter de fin de línea o de tabulación como separador.

Cuando se ejecuta, la proposición *read* hace que los valores se almacenen en el mismo orden en que se especificaron. Por ejemplo, supóngase que se declara *impuesto* como variable real y *núm* como variable entera. En ese caso, la proposición *read*

**read (impuesto, núm)**

provocaría que la computadora espere una constante real, por lo menos un espacio en blanco (o carácter de fin de línea o de tabulación) y una constante entera. Por ejemplo, si se teclean los datos

0.05        7

entonces se almacenaría el valor 0.05 en *impuesto* y el valor 7 en *núm*. Nótese que esta proposición *read* también se hubiera podido escribir como dos proposiciones *read* separadas:

**read (impuesto); read (núm)**

La otra proposición de entrada, *readln*, es similar a la proposición *read*. Para comprender perfectamente la diferencia entre las dos se requieren más antecedentes acerca de la organización de los datos en un dispositivo de entrada.

<sup>1</sup> Técnicamente, *read* y *readln* no son proposiciones, sino más bien procedimientos. En casi todos los casos, esta diferencia carece de importancia, por lo que se seguirán llamando proposiciones *read* y *readln*.

Muchas veces se trabajará con datos organizados en *líneas* o renglones. Por lo común una línea contiene un número arbitrario de caracteres (que pueden ser dígitos, puntos decimales, signos de más y menos, etc.). Puede considerarse que cada línea tiene un carácter “invisible” que marca el final de la línea, y que corresponde directamente a la tecla de retorno de carro (*return*) que se oprime al final de cada línea de entrada. Naturalmente, a este carácter se le llama *carácter de fin de línea* (que aquí se escribe `<eoln>`, de *end-of-line*). Casi siempre estos caracteres de fin de línea, así como los espacios en blanco y los caracteres de tabulación, se tratan como separadores entre las constantes reales y enteras de la entrada, pero por lo demás se hace caso omiso de ellos.

La proposición `readln` permite tener un poco más de control sobre el procesamiento de datos de entrada que tienen el formato de líneas. Específicamente, la proposición `readln` realiza exactamente las mismas funciones que la proposición `read`, con una excepción. Después de obtenerse y almacenarse un valor para cada variable de la lista de entrada, la proposición `readln` hace que la computadora deseche cualquier otra cosa que quede sin utilizarse en la última línea procesada, incluso el carácter de fin de línea. Esto implica que la siguiente proposición `read` o `readln` procesará los datos a partir del principio de la siguiente línea de entrada completa.

Como ejemplo, supóngase que los datos de entrada están organizados así:

```
5      13      17<eoln>
2      9<eoln>
```

Supóngase que se declararon *núm1*, *núm2* y *núm3* como variables enteras. Entonces, la secuencia de proposiciones de entrada

```
readln (núm1);
readln (núm2, núm3)
```

haría que se almacene el valor 5 en *núm1*. Puesto que se ejecutó una proposición `readln`, el resto de la línea se desecha. La siguiente proposición `readln` comienza a procesar los datos de la segunda línea de entrada, de manera que se almacena el valor 2 en *núm2* y el valor 9 en *núm3*.

La siguiente tabla contiene ejemplos que indican los valores que se almacenan en *núm1*, *núm2* y *núm3* después de la ejecución de proposiciones `read` y `readln`.

VALORES ASIGNADOS

Ejemplo	<i>núm1</i>	<i>núm2</i>	<i>núm3</i>
1 <code>readln (num1, num2);</code> <code>readln (num3)</code>	5	13	2
2 <code>read (num1, num2);</code> <code>readln (num3)</code>	5	13	17
3 <code>readln;</code> <code>readln (num1, num2)</code>	2	9	?
4 <code>readln (num1);</code> <code>read (num2);</code> <code>readln (num3)</code>	5	2	9

En cada caso se utilizan los datos que se mostraron anteriormente. Estúdiense con cuidado estos ejemplos y trátense de explicar las diferencias entre las proposiciones *read* y *readln*.

Como un ejemplo más, considérese la siguiente secuencia de proposiciones de entrada (supóngase que se declararon como enteras todas las variables mencionadas):

```
read (x, y);
readln (z);
readln (s, t);
read (w)
```

Supóngase que en la entrada aparecen las siguientes constantes enteras, organizadas en líneas, como se muestra:

5	6	9	3<eoln>
1	4	7	2<eoln>
8<eoln>			

La primera proposición

**read (x, y)**

haría que se asigne el valor 5 a *x* y el valor 6 a *y*. Obsérvese que todavía quedan datos en la primera línea.

La segunda proposición

**readln (z)**

hará que se asigne a *z* el valor 9 (la siguiente constante entera que todavía no se ha procesado) y *se deseche después el resto de la línea*. Esto incluye a los espacios en blanco antes del 3, el 3 mismo y el carácter de fin de línea.

La tercera proposición

**readln (s, t)**

hará que se asigne el valor 1 a *s* y el valor 4 a *t*. Puesto que se trata de una proposición *readln*, el resto de esta línea se desecha también, incluso el carácter de fin de línea.

La última proposición

**read (w)**

hará que se asigne el valor 8 a *w*. Nótese que en este caso no se ha desechado el carácter de fin de línea de la tercera línea.

Utilizar *readln* para desechar datos no es necesariamente un error. Lo normal es que se utilice *readln* cuando sea preciso comenzar a procesar datos desde el principio de una línea, por lo que se desecha todo lo que queda en la línea precedente, incluso el carácter de fin de línea. Los caracteres que se desechan suelen ser sólo espacios en blanco y el carácter de fin de linea, pero, como ya se vio, puede

tratarse de datos que de otra manera serían procesados. Es responsabilidad conjunta de la persona que prepara los datos de entrada y la persona que prepara el programa, asegurarse de que los datos de entrada y las proposiciones *read* y *readln* estén sincronizadas para prevenir la pérdida de datos útiles.

En resumen, he aquí varios puntos importantes que hay que tener en cuenta cuando se preparan proposiciones *read* y *readln* para introducir datos numéricos:

- El tipo de datos del valor que aparece en la entrada debe concordar con el tipo de datos de la variable correspondiente en la proposición *read* o *readln*. (Si no es así, se emitirá un mensaje de error apropiado.)
- Los datos numéricos de la entrada deben estar separados entre sí por lo menos por un espacio en blanco, carácter de tabulación o de fin de línea. (Se pueden utilizar más de uno, o una combinación de ellos, para separar datos.)
- La proposición *readln* hace que la parte que no se usó de la última línea procesada se deseche, incluso el carácter de fin de línea.
- Si no hay suficientes datos en la línea de datos actual para asignar valores a todas las variables especificadas en la proposición *read* y *readln*, se pasará por alto el carácter de fin de línea y se continuará la entrada con los datos de la siguiente línea.
- Si se llega al final de los datos de entrada antes de obtener un valor para todas las variables de una proposición *read* o *readln*, ocurrirá un error.

## Entrada de caracteres

*Read* y *readln* procesan datos de caracteres de manera similar a los datos numéricos. La diferencia es que los espacios en blanco y caracteres de tabulación o de fin de línea nunca se desechan cuando se va a leer una variable de carácter. En vez de ello, se lee el siguiente carácter de los datos de entrada y se asigna a la variable de carácter especificada en la lista de variables de la proposición *read* (o *readln*). Lo usual es que los espacios en blanco y caracteres de tabulación y de fin de línea se traten todos como espacios en blanco, pero en algunos sistemas los caracteres de tabulación y de fin de línea se tratan de manera diferente. Consultese la documentación de Pascal que se utiliza para obtener detalles acerca de la forma como se tratan estos caracteres.

Considérese el siguiente ejemplo en el cual todas las variables son del tipo de carácter (*char*). La proposición *read*

**read (car1, car2, car3, car4)**

leerá cuatro caracteres. Si se representa el carácter de fin de línea (normalmente invisible) por medio de *<eoln>*, entonces la línea de entrada

**12D <eoln>**

asignaría ‘1’ a *car1*, el carácter ‘2’ a *car2*, ‘D’ a *car3*, y el espacio en blanco (‘ ’) a *car4*. El carácter de fin de línea no se leería. La siguiente proposición *read* o *readln* tendría que ocuparse primero del carácter de fin de línea. Obsérvese que, a dife-

rencia de las constantes de carácter, los datos de entrada no requieren apóstrofes (y normalmente no deben incluirlos) para encerrar cada carácter.

Por otro lado, supóngase que se utilizó la siguiente proposición *readln*:

**readln (car1, car2, car3, car4)**

En este caso se asignarán exactamente los mismos caracteres a las variables, pero se desechará el carácter de fin de línea.

Por último, considérese esta proposición:

**read (car1, car2, car3, car4, car5)**

Si se utilizan los mismos datos de entrada, se asignarán los mismos caracteres a las variables *car1* a *car4*. Cuando se vaya a leer el valor para la variable *car5*, el “mecanismo” de entrada estará colocado en el carácter de fin de línea. Así, se leerá el carácter de fin de línea, lo que dejará al mecanismo de entrada en el primer carácter de la siguiente línea, y se asignará un carácter de espacio en blanco a la variable *car5*. (Algunas versiones no estándar de Pascal podrían producir un carácter o caracteres diferentes cuando se lee el fin de línea.)

### **Entrada mixta: caracteres y números**

Debe tenerse cuidado al leer variables tanto numéricas como de carácter de la misma línea. Como podría esperarse, la lista de variables de las proposiciones *read* y *readln* puede contener una combinación de variables enteras, reales y de caracteres.<sup>2</sup> Recuérdese que cuando se lee una variable numérica se desechan todos los espacios en blanco y caracteres de tabulación o de fin de linea precedentes y se lee una constante numérica. La lectura continúa mientras los caracteres subsecuentes sean partes legales de la constante numérica. El valor de la constante recién leída se asigna entonces a la variable numérica correspondiente. El primer carácter no utilizado para la constante numérica se deja para la siguiente operación de entrada.

Por ejemplo, supóngase que se va a procesar la proposición

**read (núm1, car1, núm2)**

donde *núm1* y *núm2* son del tipo entero y *car1* es del tipo ~~real~~<sup>char</sup>. Si se proporcionan los datos de entrada

172.0534

para que los procese la proposición *read*, entonces se asignará a la variable *núm1* el valor 172 (ya que el punto no puede formar parte de una constante entera), a

<sup>2</sup>Pascal permite la lectura de variables de otros tipos, pero sólo los tipos entero, real y de caracteres (no booleano) se deben procesar de una forma previamente definida. Se sugiere al lector consultar la documentación del compilador de Pascal que usa para determinar si es posible leer variables de otro tipo de datos y, en caso que así sea, cómo deben aparecer en la entrada.

*car1* se le asignará el valor de carácter '.', y a *núm2* se le asignará el valor 534. Obsérvese que si se procesaran los mismos datos de entrada con la proposición

**read (real1)**

donde *real1* es del tipo real, entonces se asignaría 172.0534 a *real1*.

## Salida

Ahora se examinarán las diversas formas de exhibir datos de salida en Pascal. Existen dos proposiciones de salida, *write* y *writeln*, que, al igual que *read* y *readln*, incluyen una lista de elementos que se van a exhibir, encerrados entre paréntesis y separados por comas.<sup>3</sup> La lista de salida puede contener expresiones (que incluyen a las variables) de los tipos entero, real, de carácter, booleano o de cadena (de caracteres).

Estúdiense el siguiente fragmento de un programa en Pascal, donde se declaró *prueba* como entero y *calif* como *char*:

```
prueba := 95;
calif := 'A';
write ('Calificación por letra ', calif, ', Calificación numérica ', prueba)
```

La salida que produce la proposición *write* podría tener este aspecto:

Calificación por letra A, Calificación numérica 95

Cada uno de los elementos (variables, expresiones o constantes) que se exhiben aparece en su propio *campo*, o grupo de columnas consecutivas. Aquí (y en toda esta sección) se da por hecho que los enteros se exhibirán siempre en un campo de diez columnas y los ceros a la izquierda se sustituirán por espacios en blanco. De hecho, esto puede variar en las diferentes versiones de Pascal, ya que el número de columnas que se utiliza por omisión para exhibir los diferentes tipos de datos (llamado *anchura de campo*) depende de la versión. Los datos del tipo *char* (de carácter) siempre se exhiben con una anchura de campo por omisión de uno. Más adelante en esta sección se verán métodos para especificar de manera explícita la anchura de campo.

Examínense las siguientes proposiciones de asignación (*r* y *s* son enteros y *t* es *char*):

```
r := 4;
s := 5;
t := 'B';
```

A continuación se muestra la salida que producirían algunas proposiciones *write*.

<sup>3</sup>Una vez más el autor se ha tomado la libertad de tratar a *write* y *writeln* como proposiciones. Al igual que *read* y *readln*, son realmente procedimientos.

<i>proposición write</i>	<i>salida</i>
write (r)	4
write (r, s)	4 5
write ('El total es', r+s)	9
write (t)	B
write ('R = ', r)	R = 4

Nótese que los valores enteros se imprimen en un campo de columnas igual a la anchura del campo y que los valores se *justifican a la derecha*. Es decir, todos los espacios que no se utilizan a la izquierda se llenan con espacios en blanco. Así, en la primera proposición *write*, hay nueve espacios en blanco antes del 4. Obsérvese además que se pueden escribir expresiones (como  $r + s$ ) en la lista de salida, como se ilustra en la tercera proposición *write*.

Cuando aparece una expresión real en la lista de salida, se exhibe una representación decimal del valor de la expresión, redondeado a un número de dígitos previamente definido. La representación decimal que se utilice será la misma que se emplee en el caso de las constantes reales. Por ejemplo, si *impuesto* es una variable real cuyo valor es 0.05 (en notación científica,  $5 \times 10^{-2}$ ) y la anchura de campo por omisión para expresiones reales es de 20 columnas, entonces la proposición

**write (impuesto)**

produciría la salida

5 . 0 0 0 0 0 0 0 0 0 0 0 e - 0 2

Obsérvese que se usan ceros a la izquierda para llenar los espacios no utilizados del campo.

Considerese el siguiente programa corto en Pascal que ilustra la proposición *write*

**PROGRAM** escribe (input, output);  
(\* Programa que exhibe calificación numérica y por letra \*)  
VAD

VAR

```
número : integer;  
calif : char;
```

BEGIN

```
número := 90;  
calif := 'A';  
write ('Aciertos:', número);  
write (' , Calificación: ', calif)
```

END.

is

La salida tendría el siguiente aspecto:

Aciertos: \_\_\_\_\_ 90. Calificación: \_\_\_\_ A

La proposición *writeln* es similar a la proposición *write* con la misma lista de salida, con la excepción de que después de escribir todos los valores se escribe además un carácter de fin de línea. Puesto que el efecto del fin de línea es comenzar un renglón nuevo, las siguientes proposiciones

```
writeln ('uno');
writeln ('dos')
```

darían lugar a la exhibición de dos renglones, y la siguiente proposición *write* o *writeln* comenzaría su salida en el tercer renglón:

```
uno
dos
```

Nótese que si se hubieran utilizado proposiciones *write* en vez de *writeln*, la salida sería bastante diferente. Se hubiera escrito:

```
unodos
```

y la siguiente proposición *write* o *writeln* continuaría colocando la salida en el mismo renglón.

Se puede utilizar *writeln* sin lista de salida para hacer que se imprima o exhiba un solo carácter de fin de línea. En este caso se omiten los paréntesis que delimitan a la lista de salida. Por ejemplo, las proposiciones

```
writeln ('uno');
writeln;
writeln ('dos');
```

producirían una salida de tres renglones, de los cuales el segundo queda en blanco:

```
uno
(Renglón en blanco.)
```

```
dos
```

He aquí otro programa corto en Pascal que ilustra la proposición *writeln*:

```
PROGRAM correcto (input, output);
VAR
  integer integer número : entero;
  númerreal : real;
BEGIN
  número := 12;
  númerreal := 12.3456;
  writeln ('El valor es', número);
  writeln ('Valor real;', númerreal)
END.
```

```
E I v a l o r _ e s ----- 1 2
V a l o r _ r e a l ----- 1 2 3 4 5 6 0 0 0 0 0 0 0 e + 0 1
```

Nótese que una secuencia larga de proposiciones *write* que no se vea interrumpida por proposiciones *writeln* puede producir renglones extremadamente largos. Muchos compiladores de Pascal tienen una longitud de línea máxima que no se puede exceder. Además, muchos dispositivos de salida tienen limitaciones inherentes en cuanto a la longitud de los renglones. Muchas terminales de TRC, por ejemplo, exhiben un máximo de 80 caracteres por línea, y la mayor parte de las impresoras son incapaces de escribir más de 132 caracteres por renglón. Si se excede esta longitud, es casi seguro que el resultado será inaceptable. Debe estudiarse minuciosamente la organización de las salidas de cada programa, y es preciso utilizar la combinación correcta de proposiciones *write* y *writeln*.

Como ya se vio en los ejemplos precedentes, es posible exhibir cadenas de caracteres. Las cadenas se encierran entre apóstrofos y se exhiben únicamente los caracteres de la cadena. La anchura de campo por omisión de esas cadenas es igual al número de caracteres de la cadena. Nótese que los espacios en blanco dentro de una cadena se toman en cuenta para determinar la longitud de campo por omisión.

También es posible incluir expresiones booleanas en la lista de salida de una proposición *write* o *writeln*. El valor de la expresión, *true* o *false*, se maneja como cadena y es esta cadena la que se exhibe. (El que se exhiban letras mayúsculas o minúsculas dependerá del compilador utilizado.) Nótese que la longitud de campo por omisión es entonces de cuatro o cinco caracteres, respectivamente. Las siguientes proposiciones ilustran la salida de valores booleanos.

<i>proposición</i>	<i>datos</i>
<i>write (true)</i>	T R U E
<i>write (false)</i>	F A L S E

## Salidas con formato

Si se emplean proposiciones *write* o *writeln* en la forma arriba descrita (es decir, si se utilizan las anchuras de campo por omisión elegidas por el creador de la versión de Pascal usada), no se puede controlar el número de columnas que se van a utilizar para expresiones numéricas. Además, son obligatorias las anchuras de campo por omisión especificadas por Pascal para los tipos de datos de caracteres, de cadena y booleanos. Si bien estas anchuras de campo por omisión pueden ser adecuadas para algunas aplicaciones, el programador necesitará muchas veces

controlar con mayor precisión la colocación de valores en los renglones de las salidas.

Pascal permite al programador especificar de manera explícita las anchuras de campo para cada expresión que se va a exhibir. Estas anchuras explícitas anulan a las anchuras de campo por omisión que se usarían normalmente. Para ver cómo se especifican estas anchuras de campo, examíñese el siguiente segmento de programa. Todas las variables empleadas se declararon como enteras.

```
núm1 := 4;
núm2 := -12;
write (núm1:5, núm2:7, 'Nada' :6)
```

Estas proposiciones producirían la siguiente salida:

```
-----4 -----1 2 ___N a d a
```

Cada una de las expresiones de la lista de salida va seguida de un signo de dos puntos y una expresión entera (en este ejemplo, una constante entera) que especifica la anchura de campo que se desea utilizar para exhibir el valor de la expresión. Al igual que en el caso de las anchuras de campo por omisión, los valores se justifican a la derecha en el campo y se usan espacios en blanco a la ~~derecha~~<sup>izquierda</sup> para llenar el campo hasta la anchura especificada. Esta técnica (llenar un campo con espacios a la derecha) se utiliza para todos los tipos de datos cuando la anchura de campo explícita especificada es mayor de lo necesario. Si la anchura de campo explícita queda “justa”, no aparecen espacios en blanco a la derecha

### Anchuras de campo explícitas para expresiones enteras

Pero puede uno preguntarse qué sucedería si la anchura de campo que se especifica es demasiado pequeña para exhibir correctamente el valor de una expresión entera. Considérese el caso en que se especifica una anchura de campo de dos para el valor  $-12$ . Recuérdese que es necesario exhibir también el signo menos.

writeln ( $-12:2$ )

Podría suponerse que se omitirá parte del número, pero esto quizás provocará problemas serios en la interpretación de las salidas. Por ejemplo, podría suponerse que se omiten los caracteres sobrantes a la derecha, en cuyo caso el valor  $-12$  se exhibiría como si fuera el valor  $12$ . Sin duda, esto engañaría a quien tratara de utilizar esta salida.

En vez de truncar caracteres en la exhibición de valores enteros cuando la anchura del campo es insuficiente, Pascal “ensancha” automáticamente el campo para que se pueda exhibir correctamente el valor. En el último ejemplo, se haría caso omiso de la anchura de dos y se exhibiría el valor  $-12$  en un campo de tres columnas. Este ensanchamiento de los campos también se hace en el caso de los valores reales. Los campos para valores reales se analizarán con mayor detalle más adelante.

En las listas de salida se puede tratar una expresión del tipo *char* exactamente como si fuera una cadena de un carácter. Recuérdese también que los valores booleanos en una lista de salida se tratan como la cadena correspondiente, ‘TRUE’ o ‘FALSE’.

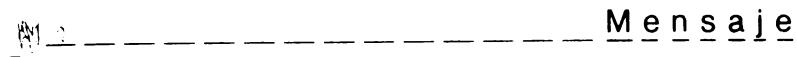
Pascal nunca modifica la anchura de campo especificada explícitamente para una cadena. Si la anchura es excesiva, se agregan espacios en blanco a la derecha para llenar el campo. Si los campos son demasiado pequeños para exhibir todos los caracteres de la cadena, se exhibirán únicamente los caracteres de la izquierda de la cadena hasta llenar el campo; los caracteres de la derecha se truncarán. Los siguientes ejemplos ilustran estos puntos:

<i>proposición</i>	<i>salida</i>	<i>observaciones</i>
write ('Mensaje')	M e n s a j e	La anchura por omisión es la longitud
write ('Mensaje':9)	M e n s a j e	Espacios en blanco a la derecha.
write ('Mensaje':4)	M e n s	Sólo se escriben cuatro caracteres.

Muchas veces es útil incluir la constante de carácter “ ” en una lista de salida con anchura de campo mayor de uno para obligar a la alineación del siguiente campo con una columna específica. Considérese, por ejemplo, la siguiente proposición:

`write (":15, 'Mensaje')`

Si se supone que se va a escribir un renglón nuevo, se observaría lo siguiente:

A hand-drawn diagram showing a horizontal line with a wavy underline underneath it, followed by the text 'Mensaje'.

### **Anchura de campo explícitas para expresiones reales**

Las expresiones reales también pueden incluir una anchura de campo explícita. De hecho, existen dos formas que se pueden emplear para la salida de valores reales con formato:

- La expresión real puede ir seguida de un signo de dos puntos y una anchura de campo (como en el caso de los enteros y caracteres):

`write (númreal:10)`

- La expresión real puede ir seguida de un signo de dos puntos y una anchura de campo y otro signo de dos puntos y una expresión (entera) que especifique el número total de dígitos fraccionarios que deben aparecer después del punto decimal:

`write (númreal:10:5)`

En la primera forma, el valor de la expresión real se escribe en la forma exponencial que ya conoce el lector (notación científica). La exhibición incluye:

- El carácter del signo (un signo menos si es negativo, un espacio en blanco si es positivo)
- El primer dígito significativo del valor de la expresión
- El punto decimal
- Certo número de dígitos después del punto decimal (esto depende, por supuesto, de la anchura de campo especificada)
- La letra *e* o *E* (el uso de minúsculas o mayúsculas depende de la versión de Pascal)
- El signo del exponente (“+” o “-”)
- El exponente (por lo común dos o tres dígitos decimales)

Obsérvese que la salida exacta que resulte del uso de una sola anchura de campo para expresiones reales depende hasta cierto punto del compilador de Pascal que se utilice. El siguiente programa se compiló con un sistema de cómputo muy común y produjo la salida que se muestra:

<i>programa</i>	<i>salida</i>
<pre>PROGRAM demoreal (input, output); BEGIN   writeln (12345.67 : 8);   writeln (12345.67 : 9);   writeln (12345.67 : 10) END.</pre>	$\begin{array}{r} -1 \cdot 2e+04 \\ -1 \cdot 23e+04 \\ -1 \cdot 235e+04 \end{array}$

Puede verse que la salida siempre tiene exponente, y que la parte fraccionaria se *redondea* por adición de 5 al primer dígito fraccional que no se exhibe. El número real de columnas utilizado (esencialmente, la anchura de campo mínima) es por lo menos seis más que el número de dígitos que tiene el exponente. Nótese también que no se agregan espacios en blanco adicionales a la izquierda (a excepción del que representa al signo + si el número es positivo) cuando se emplea este formato. Las columnas adicionales se llenan con dígitos después del punto decimal. En el

sistema que se utilizó aquí, por ejemplo, el número mínimo de columnas utilizado es seis, ya que el exponente siempre requiere dos dígitos decimales.

La segunda forma de salida real con formato produce una representación de **punto fijo**; es decir, no se exhibe el exponente y el número aparece en su formato decimal. La expresión después del primer signo de dos puntos, como siempre, determina la anchura mínima de campo que se usa para la salida. La expresión después del segundo signo de dos puntos determina el número de dígitos que se escriben después del punto decimal. He aquí un programa similar al ejemplo anterior, con la salida correspondiente:

<i>programa</i>	<i>salida</i>
<pre>PROGRAM demoreal2 (output); BEGIN   writeln (12345.67 : 9 : 2);   writeln (12345.67 : 10 : 3);   writeln (12345.67 : 11 : 2)</pre>	$\begin{array}{ccccccccc} & 1 & 2 & 3 & 4 & 5 & . & 6 & 7 \\ - & \underline{1} & \underline{2} & \underline{3} & \underline{4} & \underline{5} & . & \underline{6} & \underline{7} \\ - & & & & & &   & & 0 \\ - & & & & & & \underline{1} & \underline{2} & \underline{3} \\ & & & & & & \underline{4} & \underline{5} & \underline{6} \\ & & & & & & & \underline{7} & \end{array}$

En la representación de punto fijo se usan espacios en blanco a la izquierda para que el campo tenga la anchura especificada. (Examíñese la tercera proposición *writeln*.) Además, siempre se dedicará al signo una columna que esté antes del primer dígito (un signo menos si es negativo, un espacio en blanco si es positivo). Trate el lector de explicar por qué se obtuvo la salida indicada con la segunda proposición *writeln*.

## EJERCICIOS DE LA SECCIÓN 3.1

- 1 Supóngase que se tiene la siguiente secuencia de proposiciones de entrada (considérese que todas las variables se declararon como enteras):

```
read (x, y, z);
readln (a);
readln (b, c);
read (d)
```

Obtégase los valores de las variables (*a*, *b*, *c*, *d*, *x*, *y* y *z*) si se utilizaron estos datos de entrada:

8	7	2	1	3
1	4	4	6	2
3	7			

- 2 Dadas las siguientes declaraciones de variables

**VAR**

```
a, b, c : integer;
x, y, z : real;
```

y los siguientes datos de entrada

3	2.3	–6.5
1	5	2.1

encuéntrense los errores, si existen, en las siguientes proposiciones *read*.

- (a) `read (x, y, z)`
  - (b) `read (a, b, x)`
  - (c) `read (a, x, y);  
read (b, z, c)`
  - (d) `readln (a, x);  
read (b, c, y, z)`
- 3 ¿Qué se exhibirá (en el dispositivo de salida) cuando se ejecuten las siguientes proposiciones en Pascal?
- ```
writeln ('Valor1 es', 3);
writeln ('Valor2 es', 5);
writeln ('La suma es' 3 + 5)
```
- 4 ¿Qué se exhibirá cuando se ejecuten las siguientes proposiciones en Pascal?

```
writeln (86, 39);
writeln ('a = ');
writeln (32.5);
writeln ('a = ', 86, 'b = ', 32.5)
```

- 5 ¿Qué se exhibirá cuando se ejecute el siguiente programa en Pascal?

```
PROGRAM adivina (input, output);
VAR
    a, b, c : integer;
    x, y, z : real;
BEGIN
    a := 0;
    b := 2;
    c := 1;
    x := 5.2;
    y := 3.6;
    z := 4.1;

    write ('Los valores son');
    write (a, b, c,);
    writeln (x, y, z);
    writeln ('La suma es', x + y + z);
    writeln ('El producto es', a * b * c)
END.
```

- 6** Determíñese la salida exacta del siguiente programa en Pascal.

```

PROGRAM salea (input, output);
VAR
  a : char;
  b : integer;
BEGIN
  a := 'x';
  b := -12;
  writeln (a:2, b:5)
END.
  
```

- 7** Supóngase que  $a$ ,  $b$ ,  $c$  y  $d$  son variables enteras. Determíñense los valores de  $a$ ,  $b$ ,  $c$  y  $d$  después de la ejecución de las siguientes proposiciones con los datos que se muestran.

| <i>proposición</i> | <i>datos</i> |    |    |   |
|--------------------|--------------|----|----|---|
| readln (a, b);     | 2            | 15 | 6  | 4 |
| readln (c);        | 1            | -3 | 7  | 9 |
| read (d)           | 8            | 12 | -1 | 5 |

- 8** Supóngase que  $m$  y  $t$  son variables enteras con los valores 4 y -18, respectivamente. Determíñese la salida *exacta* de la siguiente proposición.

**write ('Valor':8,, m:2, t:4)**

- 9** Supóngase que  $a$ ,  $b$  y  $c$  son variables enteras. Un usuario escribe la siguiente secuencia de caracteres en la terminal (donde  $b$  representa un espacio en blanco):

1  $\sharp$  2

3  $\sharp$  4

¿Cuál proposición o grupo de proposiciones no producirá los valores  $a = 1$ ,  $b = 2$  y  $c = 3$ ?

- (a) read (a);  
readln (b);  
read (c)
- (b) readln (a, b, c)
- (c) read (a, b, c)
- (d) read (a);  
read (b);  
readln;  
read (c)

(e) `readln (a);  
readln (b);  
readln (c)`

- 10** ¿Cuáles de las siguientes proposiciones harán que se escriba la palabra *PRI-MERA* en las columnas 1 a 7, el valor de la variable entera siguiente de manera que termine en la columna 20, y que se escriba la palabra *ÚLTIMA* en las columnas 75 a 80?

(a) `write ('PRIMERA', siguiente:13, 'ÚLTIMA':60)`  
 (b) `write ('PRIMERA':7, siguiente:20, 'ÚLTIMA':80)`  
 (c) `write ('PRIMERA':7, SIGUIENTE:13, 'ÚLTIMA':60)`  
 (d) `write ('PRIMERA': SIGUIENTE:20, 'ÚLTIMA':80)`  
 (e) `write ('PRIMERA':7, siguiente:5, 'ÚLTIMA':56)`

- 11** Supóngase que  $x$ ,  $y$  y  $z$  son variables enteras. Dado el siguiente segmento de programa

```
readln (x);
readln (y);
readln (z)
```

¿qué se almacenará en  $x$ ,  $y$  y  $z$  si un usuario escribe las siguientes líneas cuando se ejecuta el segmento de programa?

```
1 2 ♂ 3 4 ♂ 5 6
7 . 0 ♂ 9 ♂ 1 . 0
♂ ♂ ♂ ♂ ♂ ♂
♂ 1 1 ♂ 2 2
```

(Supóngase que ♂ representa un espacio en blanco tecleado.)

- 12** ¿Cuál será la salida exacta que se produce cuando se ejecuta el siguiente programa?

```
PROGRAM prueba (input, output);
CONST hey = 'HOLA';
VAR r, s : real;
BEGIN
  r := 6.1;
  s := 7.2;
  writeln (hey : 6);
  write ('R = ');
  write (r:5:2);
  writeln;
  write ('S = ');
  write (s:3:1)
END.
```

En esta sección el proceso de resolución de problemas que se bosquejó en el capítulo 1 se aplicará a problemas con soluciones sencillas. Esto permitirá usar las características del lenguaje Pascal que se han presentado hasta ahora para obtener soluciones en computadora. En capítulos subsecuentes se presentarán problemas más complejos y sus soluciones.

Considérese el siguiente problema.

### Problema 3.1

*Determinese el total y promedio (redondeado al entero más cercano) de cuatro calificaciones de examen enteras.*

Después de leer con detenimiento y de analizar el enunciado del problema, se encuentra una solución sencilla. El problema se divide en cuatro subproblemas distintos (véase la Fig. 3-1):

SUBPROBLEMA 1: Obtener las calificaciones de examen de la entrada.

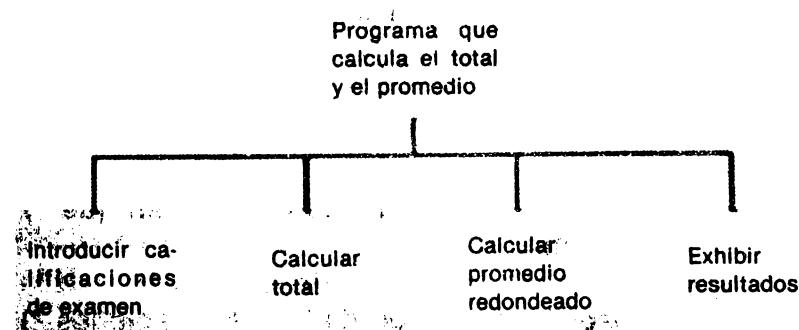
SUBPROBLEMA 2: Calcular el total de las calificaciones de examen.

SUBPROBLEMA 3: Calcular el promedio redondeado.

SUBPROBLEMA 4: Exhibir los resultados.

Nótese que la entrada y salida de los datos no se incluyó de manera específica en el enunciado del problema. El lector encontrará muchas veces problemas que están escritos de tal manera que se puede inferir la presencia de valores para "datos supuestos" (en este caso, las cuatro calificaciones de examen) y es necesario comprender cuáles de estos "datos supuestos" representan constantes y cuáles representan valores variables de los datos. Igualmente, una persona que utilice lápiz y papel para resolver un problema verá inmediatamente el resultado de los cálculos, ya que aparecen en el papel. Los resultados que se obtienen por medio de una solución en computadora tendrán que ser transferidos explícitamente de las variables y expresiones localizadas en la memoria de la computadora, al dispositivo de salida.

**Figura 3-1** Diseño descendente del programa que calcula el total y el promedio.



Es menester ahora refinar el problema de tal forma que permita la traducción inmediata a Pascal. Una refinación posible es:

- SUBPROBLEMA 1: Introducir *exam1*, *exam2*, *exam3* y *exam4*.
- SUBPROBLEMA 2: Total = *exam1* + *exam2* + *exam3* + *exam4*.
- SUBPROBLEMA 3: Promedio = redondear (total / 4).
- SUBPROBLEMA 4: Exhibir *exam1*, *exam2*, *exam3*, *exam4*, *total*, *promedio*.

Definitivamente, esto no es Pascal, pero las proposiciones son lo bastante cercanas a Pascal como para permitir su traducción inmediata. Al refinar las proposiciones anteriores se han introducido detalles adicionales: los datos de entrada ya tienen nombres (*exam1*, *exam2*, *exam3* y *exam4*) y se introdujeron los conceptos algebraicos de *suma* y *promedio*. La refinación del paso 3 todavía adolece de cierta generalidad: se supone la existencia de una función para redondear un valor real al valor entero más cercano. Antes de la traducción a algunos lenguajes, sería preciso refinar aún más el concepto de redondeo, sin embargo, Pascal incluye una función que hace precisamente eso (*round*).

Ahora ya es posible traducir esta forma refinada de la solución a un programa fuente en Pascal. Aunque ya se dieron nombres a las variables que se van a usar, todavía falta escoger un nombre para el programa. Como siempre, se escoge un nombre que tenga cierto valor *nemotécnico*, o de ayuda a la memoria, como *promexam*. No hay duda de que éste sugiere al lector no informado el tema que trata el programa. La primera línea del programa, por tanto, será:

**PROGRAM promexam (input, output);**

Si bien la refinación actual de la solución no especifica los tipos de datos que se manejan, quizá hubiera sido conveniente agregar un resumen que describiera el tipo y uso de cada dato. El resumen podría tener este aspecto:

| variable           | tipo   | uso                                                        |
|--------------------|--------|------------------------------------------------------------|
| <i>exam1,exam2</i> | Entera | Las cuatro calificaciones de examen que se introducen      |
| <i>exam3,exam4</i> |        |                                                            |
| <i>total</i>       | Entera | Suma de las cuatro calificaciones                          |
| <i>promedio</i>    | Entera | Promedio redondeado de las cuatro calificaciones de examen |

Si se prepara un resumen de variables como éste para cada una de las refinaciones de los problemas, ya se habrá preparado muy bien el camino para producir las declaraciones de variables del programa en Pascal. Obsérvese la gran similitud entre las declaraciones en Pascal y el resumen de variables:

|                               |                                                 |
|-------------------------------|-------------------------------------------------|
| <b>VAR</b>                    |                                                 |
| <i>exam1,exam2,</i>           | (* Las cuatro calificaciones de examen que *)   |
| <i>exam3,exam4 : integer;</i> | (* se introducen *)                             |
| <i>total : integer;</i>       | (* suma de las cuatro calificaciones *)         |
| <i>promedio : integer;</i>    | (* promedio redondeado de las calificaciones *) |

Por último, cada una de las cuatro refinaciones de los subproblemas se traducen a sus equivalentes en Pascal, sin olvidar encerrar el grupo completo de proposiciones entre las palabras BEGIN y END. El primer paso es obtener las cuatro calificaciones de examen. Se podría optar por indicar al usuario que escriba los valores apropiados por medio de un mensaje breve, y se hará eso en la traducción del paso 1:

```
write ('Escriba por favor las calificaciones: ');
readln (exam1, exam2, exam3, exam4);
```

A continuación se calcula el total de esas calificaciones:

```
total := exam1 + exam2 + exam3 + exam4;
```

En seguida se calcula el promedio mediante la función estándar de Pascal *round*. Esta función produce el valor entero más cercano al valor de la expresión real que se le proporciona. Se utiliza la operación de división real para proporcionar a *round* el valor real:

```
promedio := round (total /4);
```

Por último, se exhiben los resultados. En la refinación del paso 4 se optó por exhibir no sólo el total y promedio de las calificaciones de examen, sino también las calificaciones mismas. La exhibición de datos de entrada se denomina en ocasiones *impresión de eco*, o sencillamente *verificación de eco*, y permite al usuario de un programa verificar que los datos que se proporcionaron al programa en realidad son los correctos. Los programas interactivos pequeños como el que se estudia ahora normalmente se pueden escribir sin verificaciones de eco, ya que la computadora misma hace eco de los datos en el momento en que se escriben. Aquí se incluye la verificación de eco sólo para ilustrar la forma como se lleva a cabo.

Todos los valores de salida se escriben junto con un mensaje que los identifica. ¡De otra manera, el desafortunado usuario recibiría solamente seis valores sin tener idea de cuál es cuál!

```
writeln ('Las calificaciones de examen son:');
writeln ('      ', exam1, exam2, exam3, exam4);
writeln ('El total de las calificaciones es', total);
writeln ('El promedio redondeado de las calificaciones es', promedio)
```

A continuación se muestra el programa completo. Se incluyen comentarios adicionales delimitados por (\*) y \*) para mejorar la legibilidad del programa. Se emplean signos de punto y coma para separar las distintas proposiciones (aunque debe observarse que se omite después de la última proposición). También se muestra un posible resultado de la ejecución del programa.

```
PROGRAM promexam (input, output);
VAR
  exam1,exam2,      (* Las cuatro calificaciones de examen que *)
```

```

exam3,exam4 : integer; (* se introducen *)
total : integer;          (* suma de las cuatro calificaciones *)
promedio : integer;        (* promedio redondeado de las calificaciones *)
BEGIN
(* Obtener de la entrada las cuatro calificaciones de examen *)
write ('Escriba por favor las calificaciones: ');
readln (exam1, exam2, exam3, exam4);
(* Calcular el total y el promedio redondeado *)
total := exam1 + exam2 + exam3 + exam4;
promedio := round (total / 4);
(* Exhibir los resultados *)
writeln ('Las calificaciones de examen son:');
writeln ('      ', exam1, exam2, exam3, exam4);
writeln ('El total de las calificaciones es ', total);
writeln ('El promedio redondeado de las calificaciones es ',
promedio)
END

```

---

*Programa promexam*

Escriba por favor las calificaciones: **100 96 83 75**

Las calificaciones de examen son:

100 96 83 75

El promedio redondeado de las calificaciones es 89

*Programa promexam: ejemplo de ejecución*

Ahora se estudiará otro problema para resolución en computadora. Esta vez se utilizarán salidas con formato para hacerlas más legibles.

**Problema 3.2**

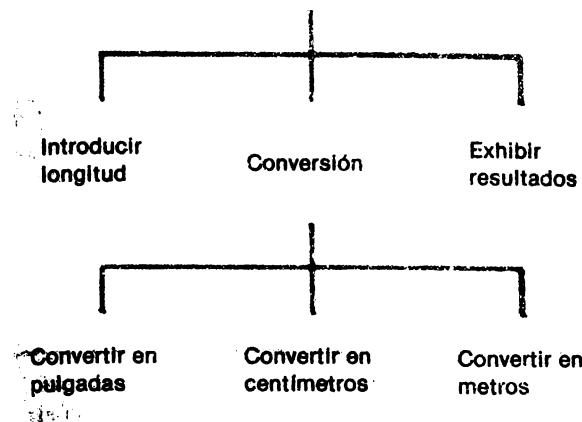
*La Compañía de Herramientas y Dados Roland está realizando una conversión del sistema inglés de medidas al sistema métrico. Escríbese un programa en Pascal que realice la siguiente tarea. Dada una longitud expresada en pies y pulgadas, determine el equivalente métrico, redondeado a dos cifras decimales, tanto en metros como en centímetros.*

Mediante el diseño descendente, este problema se puede dividir en tres subproblemas principales:

SUBPROBLEMA 1: De los datos de entrada, obtener la longitud en pies y pulgadas.  
 SUBPROBLEMA 2: Convertir la longitud en sus equivalentes métricos, primero en centímetros y después en metros.

SUBPROBLEMA 3: Exhibir los resultados de la conversión.

El subproblema de la conversión se puede subdividir en tres subproblemas (véase la Fig. 3-2):



**Figura 3-2** Diseño descendente del programa métrico.

SUBPROBLEMA 2.1: Convertir pies y pulgadas en pulgadas.

SUBPROBLEMA 2.2: Convertir pulgadas en centímetros.

SUBPROBLEMA 2.3: Convertir pulgadas en metros.

En forma de algoritmo, se tiene la siguiente secuencia de pasos:

PASO 1: Introducir la longitud.

PASO 2: Convertir la longitud en pulgadas, exclusivamente.

PASO 3: Convertir la longitud dada en pulgadas, en centímetros.

PASO 4: Convertir la longitud dada en pulgadas, en metros.

PASO 5: Exhibir los resultados.

En seguida, se refina cada uno de estos pasos para producir una forma del algoritmo más cercana a la que requiere Pascal:

PASO 1: Introducir longitud de pies y pulgadas (variables *lonpies* y *lonpuig*).

PASO 2: Convertir la longitud en pies y pulgadas a longitud en pulgadas exclusivamente (variable *sólopulg*):

*sólopulg* := *lonpies* \* 12 + *lonpulg*

(ya que un pie equivale a 12 pulgadas).

PASO 3: Convertir la longitud en pulgadas a longitud en centímetros (variable *loncm*):

*loncm* := *sólopulg* \* 2.54

(ya que una pulgada es exactamente 2.54 centímetros).

PASO 4: Convertir la longitud en pulgadas en longitud en metros (variable *lonm*):

$$\text{lonm} := \text{sólopulg} * 0.0254$$

(ya que una pulgada tiene 2.54 centímetros y un metro tiene 100 centímetros).

PASO 5: Exhibir la longitud en pies y pulgadas, metros redondeados y centímetros redondeados

He aquí un resumen de las variables que se utilizan aquí.

| variable | tipo   | uso                                   |
|----------|--------|---------------------------------------|
| lonpies  | Entera | longitud (en pies) de la entrada      |
| lonpulg  | Entera | longitud (en pulgadas) de la entrada  |
| sólopulg | Entera | longitud (en pulgadas exclusivamente) |
| lonm     | Real   | longitud (en metros)                  |
| loncm    | Real   | longitud (en centímetros)             |

El lector se preguntará quizá por qué se hizo la conversión directa de pulgadas a metros en el paso 4, en vez de convertir la longitud en centímetros (*loncm*) directamente a metros. De hecho, hay dos razones para tomar esta decisión en apariencia sin fundamento. En primer lugar, obsérvese que la especificación del subproblema indica que se convertirá la longitud en pulgadas a longitud en metros. Técnicamente, no se estaría poniendo en práctica correctamente este paso si se utilizara el resultado del paso 3 (longitud en centímetros) para determinar la longitud en metros. El lector pensará seguramente que se trata de un detalle pequeño, y lo es. Si el programador no se apega a los pequeños detalles de la especificación de subproblemas, no podrá esperar que la solución general al problema, más compleja, se comporte precisamente como se estableció durante el diseño descendente. Después de todo, el diseño descendente sólo funciona porque la solución a un problema complejo está formada por un gran número de soluciones a problemas pequeños.

La segunda razón para no utilizar el resultado previo (la longitud en centímetros) al convertir a metros es un poco más difícil de explicar en este punto del estudio de la computación. Baste con establecer que cuando se llevan a cabo cálculos numéricos es mejor, en la medida de lo posible, trabajar con los datos originales que con datos que resultan de pasos intermedios. (Considérense las siguientes analogías frívolas: al buscar un tesoro enterrado, probablemente será mejor usar el mapa original del tesoro que trazó el pirata que otro copiado del original. Cuando se le cuenta algo a una persona, la cual lo relata a otra persona, que a su vez se lo cuenta a una tercera, etc., es posible que la última persona escuche algo bastante diferente de lo que se le dijo a la primera persona).

Ahora ya se tienen los detalles necesarios para escribir un programa completo en Pascal llamado métrico, el cual se muestra en seguida. También se presenta el resultado de una ejecución del programa. Obsérvese que se utilizó la propiedad de redondeo de la salida con formato de números reales para resolver el requisito de redondeo del enunciado del problema.

### SECCIÓN 3.3 INTRODUCCIÓN A LA ENTRADA Y SALIDA DE ARCHIVOS DE TEXTO [OPCIONAL]

En un encabezado de programa como éste:

```
PROGRAM triángulo (input, output);
```

los identificadores *input* y *output* se llaman *variables de archivo* y se usan cuando se desea transferir información entre la memoria y archivos externos. Lo normal es que *input* y *output* correspondan a la terminal en un sistema interactivo de cómputo. *Input* y *output* se consideran archivos estándar en Pascal. En ocasiones es necesario leer datos de otros archivos externos (grabados en disco o cinta magnética) o quizás grabar los resultados en un archivo externo. Pascal cuenta con un mecanismo para manejar tales entradas y salidas de archivos externos.

Todo archivo externo que se emplee en un programa debe contar con las variables de archivo correspondientes. Éstas se deben listar, por nombre, en el encabezado del programa y también deben declararse como de tipo *text* (texto) en la parte de declaraciones del programa. Cuando se hace esto, se indica que el archivo está organizado por líneas, cada una de las cuales contiene cero o más caracteres y un carácter de fin de línea (*<eoln>*). Los archivos estándar *input* y *output* se declaran automáticamente como de tipo texto cuando se listan en el encabezado del programa. Supóngase que se desea utilizar dos archivos de texto externos, uno para entrada y otro para salida. Se podría recurrir a las variables de archivo llamadas *entrada* y *salida* para entrada y salida, respectivamente. La siguiente declaración para un programa llamado *proceso* permite utilizar los archivos de texto externos:

```
PROGRAM proceso (entrada, salida);
```

**VAR**

```
    entrada, salida : text;
```

A continuación es preciso poder leer y grabar estos archivos en forma similar a como se hizo con los archivos estándar *input* y *output*. Las proposiciones *read*, *readln*, *write* y *writeln* con que cuenta Pascal emplean automáticamente los archivos *input* y *output*, a menos que se especifique un nombre de archivo diferente en la proposición. Por ejemplo, para leer un dato del archivo de texto externo asociado a la variable de archivo *entrada*, y asignarlo a una variable entera llamada *número*, se usaría la proposición

```
read (entrada, número)
```

Esta proposición ordena a la computadora que lea un valor entero del archivo de texto externo asociado a la variable de archivo *entrada* y que almacene el valor resultante en la variable *número*. Puesto que *entrada* se declaró como de texto, se va a leer una cadena de caracteres que represente el valor de un entero. Esta lectu-

ra se hace *exactamente* con el mismo proceso que se sigue cuando se introducen datos desde la terminal, usando el archivo estándar input.

Un paso importante para la entrada y salida de archivos es la apertura del archivo. Si se va a leer un archivo externo, es preciso abrirlo, o conectarlo a la variable de archivo asociada, antes de poder comenzar a leer. En Pascal, la proposición *reset* realiza esta operación de apertura. En el problema actual, la proposición apropiada sería

### **reset (entrada)**

Cuando se ejecuta la proposición *reset*, el sistema de cómputo localizará el archivo externo apropiado (si no se puede localizar, habrá un error de ejecución) y se organizará de manera que todas las proposiciones *read* o *readln* que hagan referencia a la variable de archivo *entrada* procesen el archivo externo correspondiente, a partir de su inicio.

De manera similar, la proposición *rewrite*

### **rewrite (salida)**

conectará la variable de archivo *salida* a un archivo externo nuevo. (Si el archivo externo ya existe, se borrará su contenido previo; en caso contrario, se creará un archivo externo nuevo. La imposibilidad de crear este archivo generará un error de ejecución.) Este archivo nuevo inicialmente está vacío. Es preciso usar *rewrite* antes de ejecutar cualquier proposición *write* o *writeln* que haga referencia a *salida*. Después del *rewrite* se puede ejecutar la proposición

### **write (salida, número)**

para grabar en *salida* exactamente los mismos caracteres que se habrían exhibido en la terminal si se hubiera ejecutado la proposición

### **write (número)**

El archivo externo exacto que se asocia a una variable de archivo cuando se ejecuta un *reset* o *rewrite* varía según el sistema de cómputo que se utilice. En muchos sistemas, el *nombre de archivo* del archivo externo es el mismo que el nombre de la variable de archivo, pero pueden existir mecanismos que permitan conectar una variable de archivo con un archivo externo arbitrario. Algunos compiladores de Pascal también tienen formas ligeramente diferentes para *reset* y *rewrite* (u otras proposiciones) que permiten al programador especificar el nombre del archivo externo que se debe conectar a la variable de archivo. Se sugiere al lector consultar en la documentación de su Pascal los detalles de estas deviaciones con respecto al Pascal estándar.

Supóngase ahora que se desea escribir un programa en Pascal llamado *copiar* que lee tres valores reales de un archivo de texto externo y los copia en un archivo de texto externo diferente. Se supondrá que se usan las variables de archivo *entradas* y *saldatos*. He aquí una forma de lograrlo.

```

PROGRAM copiar (entdatos, saldatos);
(* Programa que copia tres números reales de la variable de archivo *)
(* entdatos a la variable de archivo saldatos *)
VAR
    num1, num2, num3 : real;           (* datos reales *)
    entdatos, saldatos : text;        (* variables de archivo *)
BEGIN
    reset (entdatos);                (* "abrir" entdatos *)
    rewrite (saldatos);              (* "abrir" saldatos *)
    read (entdatos, num1, num2, num3); (* obtener tres números *)
    write (saldatos, num1, num2, num3); (* grabar tres números *)
END.

```

En el capítulo 12 se volverá a hablar de las variables de archivo, pero aquí se comentan algunos puntos importantes que conviene recordar ahora acerca de la entrada y salida de archivos de texto.

- Es posible especificar más de un archivo de entrada o salida en el encabezado del programa.
- *Input* y *output* se declaran automáticamente como variables de archivo de tipo texto si aparecen en el encabezado del programa y se abren automáticamente para lectura y grabación, respectivamente.
- Todas las demás variables de archivo que aparezcan en el encabezado del programa se deben declarar como de tipo texto en la parte de declaraciones de variables del programa.
- La lectura o grabación de archivos de texto externos se hace mediante proposiciones *read*, *readln*, *write* y *writeln* que especifican una variable de archivo como primer nombre de la lista de parámetros.
- Antes de ejecutar la primera proposición *read* o *readln* que utilice una variable de archivo diferente de *input*, es necesario abrir esa variable de archivo mediante la proposición *reset*.
- Antes de ejecutar la primera proposición *write* o *writeln* que utilice una variable de archivo diferente de *output*, es necesario abrir esa variable de archivo mediante la proposición *rewrite*.

## SECCIÓN 3.4 TÉCNICAS DE PRUEBA Y DEPURACIÓN

En esta sección se examinarán algunos errores comunes que se presentan cuando se realizan operaciones de entrada o salida de datos.

### Errores de entrada

Un error común cuando se introducen datos es escribir un valor cuyo tipo no concuerda con el de la variable que se lee. Por ejemplo, si la variable *número* se declara como entera, la proposición *read*

requiere que el siguiente elemento en la entrada sea una constante entera. Si se escribe un carácter que no sea un signo (+ o —) o un dígito (0 a 9), se indicará un error de entrada. En un capítulo posterior se analizarán técnicas de validación de datos de entrada para poder detectar estos errores y emprender acciones correctivas. La verificación de errores de entrada es una parte importante de la prueba y depuración.

Otro error común de entrada es carecer de datos suficientes para satisfacer la proposición *read* o *readln* que está ejecutando. Por ejemplo, si *núm1*, *núm2*, *núm3* y *núm4* son variables enteras, la proposición

**read (núm1, núm2, núm3, núm4)**

requiere cuatro constantes enteras en los datos de entrada. Si sólo se proporcionan tres valores, se presentará un error. Si la entrada incluye otros valores (destinados a otras proposiciones *read* y *readln*), entonces se leerá (incorrectamente) el primero de éstos para asignarlo a *núm4*, lo que posiblemente provocará un resultado erróneo en los cálculos. Si el archivo termina después del valor para *núm3*, se señalará inmediatamente que hay un error.

El orden de los datos de entrada es importante. Por ejemplo, supóngase que *calif* es una variable de carácter, *prueba* una variable entera y *promedio* una variable real. La proposición

**readln (calif, prueba, promedio)**

requerirá datos de entrada parecidos a éstos:

A 95 82.6

Cuando se ejecute la proposición *read*, *calif* tendrá el valor A, *prueba* 95 y *promedio* 82.6. Recuérdese que se emplean espacios en blanco para separar valores reales y enteros. Supóngase que se modifica el orden de las variables, como en la proposición *readln*

**readln (promedio, calif, prueba)**

Si se introdujeren después los datos

82.6 A 95

entonces se asignará el valor 82.6 a *promedio*. A continuación se espera un valor de carácter, y el “apuntador de entrada” está colocado en el espacio en blanco que separa al valor 82.6 del carácter A. Este carácter (‘’) será el que se asigne a la variable *calif*. Después de esto, el apuntador de entrada estará colocado en la letra A. La proposición *readln* intentará obtener un valor entero para *prueba* que comience con el carácter A, por lo que naturalmente fracasará. Recuérdese que sólo se pasan por alto los espacios en blanco y caracteres de tabulación o fin de línea

antes de leer valores enteros o reales, y no se pasa por alto carácter alguno, ni siquiera los caracteres de fin de línea, antes de leer un valor de carácter.

El último error potencial de entrada que se presentará se refiere al uso de los apóstrofos para encerrar datos de carácter. En una expresión que incluya constantes de carácter, como en la proposición de asignación

```
calif := 'A'
```

es preciso encerrar el carácter entre apóstrofos para crear una constante de carácter válida. Pero no se deben usar estos apóstrofos cuando el mismo carácter aparece en los datos de entrada. Por ejemplo, la proposición

```
read (calif)
```

tomará el siguiente carácter de la entrada como valor para *calif*. Si los datos de entrada contienen

```
'A'
```

entonces se asignará el carácter apóstrofo a *calif*, no la letra A.

### **Errores de salida**

Un error de salida común es olvidar los apóstrofos que deben encerrar a cualquier texto que se deseé exhibir. Por ejemplo, la proposición

```
writeln (El número es, número)
```

es errónea, ya que faltan apóstrofos a ambos lados de

**El número es**

Cuando se utiliza salida con formato, las anchuras de campo deben ser valores enteros. La proposición

```
writeln (3.5 + 3.6 : 3.0)
```

es errónea porque la anchura de campo no es entera. La proposición debería ser

```
writeln (3.5 + 3.6 : 3)
```

Existen muchos tipos de errores de salida que pueden presentarse a causa de errores de sintaxis como los que ya se ilustraron. Para adquirir práctica en la localización de errores que resultan de la especificación de formatos no apropiados para los datos de salida, se recomienda al lector experimentar con las distintas formas de salida de datos en su sistema. Incluyanse proposiciones *write* como impresión de eco para validar los datos de entrada si surgen errores de entrada

y/o salida. Si se realizan cálculos aritméticos, inclúyanse proposiciones *write* para exhibir resultados intermedios. Estas proposiciones *write* de depuración se pueden eliminar sin problema una vez que se haya probado y depurado el programa.

Como ayuda para probar y depurar programas se proporcionan en seguida una lista de recordatorios importantes de Pascal para entrada y salida.

## RECORDATORIOS DE PASCAL

### *Entrada*

- Los valores enteros o reales deben estar separados por espacios en blanco (o caracteres de tabulación o de fin de línea)
- La introducción de variables de tipos de dato mixtos debe manejarse con cuidado, ya que el orden es importante
- El uso de *readln* hace que la siguiente proposición *read* o *readln* comience a procesar la siguiente línea de datos de entrada

### *Salida*

- Se puede especificar el formato de valores enteros al agregar un signo de dos puntos y una anchura de campo entera:

**write (enúmero:5)**

- Se puede especificar el formato de valores reales agregando un signo de dos puntos, una anchura de campo entera, otro signo de dos puntos y un entero que especifique el número de cifras decimales después del punto decimal:

**write (rnúmero:10:3)**

- El Pascal ampliará automáticamente la anchura de campo si no es suficiente para representar correctamente un valor; por ejemplo,

**write (enúmero:0)**

siempre utilizará un campo de una columna por lo menos

## SECCIÓN 3.5 REPASO DEL CAPÍTULO

En este capítulo se analizaron los detalles de la entrada y salida de datos en Pascal. También se incluyó una sección opcional que trató de la entrada y salida de archivos de texto. Se resolvieron completamente dos problemas mediante la estrategia de diseño descendente para resolución de problemas. En la sección de técnicas de prueba y depuración se presentaron errores comunes en la entrada y salida de datos.

He aquí un resumen de los detalles sobre entrada y salida de datos que se analizaron en este capítulo.

**REFERENCIAS DE PASCAL****1 Entrada/salida**

- 1.1 *Entrada:* para obtener datos de un dispositivo de entrada se puede utilizar la palabra *read* o *readln* seguida de una lista de identificadores de variables entre paréntesis:

```
read (número1, número2);
readln (capital, interés, impto);
```

*Readln* hace que todos los datos que sobran en la línea de datos de entrada se desechen después de leer los valores para las variables especificadas en la lista.

- 1.2 *Salida:* para exhibir datos en un dispositivo de salida se puede emplear la palabra *write* o *writeln* seguida de una lista de identificadores, texto (encerrado entre apóstrofos) o expresiones, encerradas entre paréntesis:

```
write (suma, producto);
write ('El valor es ', valor);
writeln ('El doble del valor dado es ', 2 * valor);
```

*Writeln* hace que la siguiente proposición *write* o *writeln* exhiba datos a partir del siguiente renglón.

*Salida con formato:* en las proposiciones *write* o *writeln*, al incluir un dato (identificador, texto entre apóstrofos o expresión) seguido de un signo de dos puntos y un entero (y otro signo de dos puntos opcional seguido de un entero en el caso de valores reales) se cambia la anchura de los campos por omisión:

```
writeln (número:5);
write ('El promedio es':15, prom:6:2);
```

Para los valores reales, el entero que sigue al segundo signo de dos puntos indica el número de dígitos fraccionarios que se van a exhibir.

**Avance del capítulo 4**

Después de completar este capítulo, el lector deberá ser capaz de escribir programas completos en Pascal para resolver algunos problemas básicos. Para resolver problemas más difíciles, se aplicará la estrategia de diseño descendente para dividir el problema en subproblemas y seguir refinando los subproblemas hasta que sea posible la resolución en computadora. Para poner en práctica esta estrategia mediante Pascal, se presentarán segmentos de programas llamados procedimientos que resuelven cada subproblema. Los procedimientos son en realidad subprogramas incluidos dentro del programa principal. Cada procedimiento deberá ser un subprograma autosuficiente e independiente que resuelva uno de los

### Palabra clave del capítulo 3

|                   |                     |
|-------------------|---------------------|
| archivo           | <i>readln</i>       |
| entrada           | <i>reset</i>        |
| formato           | <i>rewrite</i>      |
| impresión de eco  | variable de archivo |
| nombre de archivo | <i>write</i>        |
| <i>read</i>       | <i>writeln</i>      |

## EJERCICIOS DEL CAPÍTULO 3

### ★ EJERCICIOS ESENCIALES

- 1 ¿Cómo maneja el sistema del lector los caracteres de tabulación en la entrada de datos? Para averiguarlo, ejecútese el siguiente programa y proporcione un carácter de tabulación, un punto y un retorno de carro como datos de entrada.

```
PROGRAM tab (input, output);
VAR c1, c2, c3 : char;
BEGIN
  read (c1, c2, c3);
  writeln (ord(c1), ord(c2), ord(c3))
END.
```

- 2 ¿Qué salida producirá el siguiente programa cuando se teclee la línea de datos de entrada

#### 1.2.3.4

```
PROGRAM entraro (input), output);
```

```
  c : char;
  r : real;
  e : integer;
BEGIN
  readln (r, c, e);
  writeln (r, c, e)
END.
```

- 3 ¿Qué exhibirán las siguientes proposiciones (suponiendo que *r* es una variable real)?

```
r := 91.2;
writeln (r:3:0)
```

### ★ ★ EJERCICIOS IMPORTANTES

- 4 Las *proposiciones read* y *readln* ocultan muy bien la verdadera identidad del carácter de fin de línea. ¿Puede el lector determinar la forma como se separan las líneas individuales en su sistema de cómputo?
- 5 ¿Permite el Pascal del lector variables booleanas en proposiciones *read*? Si es así, ¿qué se debe escribir para proporcionar los datos *true* y *false*?

### ★ ★ ★ EJERCICIOS ESTIMULANTES

- 6 Determinese la longitud de la línea más larga que se puede exhibir en el sistema que se usa. Tómese en cuenta que ésta no es la anchura de la terminal. (Es posible que algunos sistemas no tengan una longitud máxima de línea.)

## PROBLEMAS DEL CAPÍTULO 3 PARA RESOLUCIÓN EN COMPUTADORA

### ★ PROBLEMAS ESENCIALES

- 1 En los datos de entrada se proporcionan dos tiempos como enteros de la forma *hhmm* donde *hh* representa las horas (menos de 24) y *mm* los minutos (menos de 60). Determinese la suma de estos dos tiempos y exhibase en la forma *d hhmm*, donde *d* es días (ya sea cero o uno).

Ejemplo de entrada:

1345        2153

Ejemplo de salida:

1 1138

- 2 El error relativo en una medición *m* es la razón de la diferencia absoluta entre la medición y el valor verdadero *v* al valor verdadero. Supóngase que los datos de entrada se dan en una sola línea que incluye dos números reales que representan la medición *m* y el valor verdadero *v*. Calcúlese el error relativo de la medición y exhibase junto con la medición y el valor verdadero. Inclúyanse etiquetas descriptivas para los valores exhibidos.

Ejemplo de entrada

51.0        51.3

Medición = 5.100000000e + 01  
 Valor verdadero = 5.130000000e + 01  
 Error relativo = 5.8479532161e -03

-  3 No es posible utilizar una computadora para generar números aleatorios genuinos porque es preciso utilizar un algoritmo para generar los números, lo que implica que es posible predecir los números generados. Lo que si pueden hacer las computadoras es generar números seudoaleatorios (números que, estadísticamente, parecen ser aleatorios). Una técnica antigua que no produce buenos resultados se llama método del *cuadrado medio*. Funciona así: dado un número  $a$ , para generar el siguiente número de la secuencia se extraen los dígitos que están en medio de  $a^2$ . Por ejemplo, si  $a$  es 53, entonces  $a^2$  es 2809, y el siguiente número seudoaleatorio será 80. Si se continúa, se ve que  $80^2$  es 6400, por lo que el siguiente número seudoaleatorio es 40. Si se continua este proceso se obtiene 60, 60, 60,.... Escribase un programa en Pascal que lea un entero de dos dígitos y determine el siguiente número seudoaleatorio que se generaría si se usara el método del cuadrado medio. Supóngase que la entrada consta de una sola linea que contiene al entero de dos dígitos. Exhibase el número de dos dígitos original, el cuadrado de este entero; y el siguiente número, todos con etiquetas apropiadas.

Ejemplo de entrada:

53

Ejemplo de salida:

Número introducido = 53  
 Cuadrado del número = 2809  
 Siguiente número seudoaleatorio = 80

- 4 Escribase un programa en Pascal que lea dos enteros que representen el peso de un objeto en libras y onzas. En seguida exhibase el peso introducido y su equivalente en kilogramos en una forma similar a la que se muestra a continuación. Una libra tiene 16 onzas y 2.2046 libras equivalen a un kilogramo.

Entrada

5 3 (que representan 5 libras y 3 onzas)

Salida:

Un peso de 5 libras y 3 onzas equivale a 2.353 kilogramos.

- 5 Un solenoide es una bobina de alambre enrollado en forma apretada con un determinado radio y longitud. Una característica eléctrica de los solenoides

es su inductancia, que está determinada por su longitud, el área de la sección transversal y el número de vueltas por unidad de longitud. La fórmula exacta es:

$$L = \mu \cdot \text{longitud} \cdot n^2 \cdot A$$

donde     $L$  = inductancia en henries  
 $\mu$  = constante de permeabilidad,  $4 \cdot \pi \cdot 10^{-7}$   
 $\text{longitud}$  = longitud del solenoide en metros  
 $n$  = número de vueltas de alambre por unidad de longitud  
 $A$  = área de la sección transversal en metros cuadrados

Escríbase un programa que obtenga  $n$ , la longitud (en pulgadas) y el radio del solenoide de los datos de entrada, y exhiba la inductancia (en microhenries). Recuérdese que una pulgada es 2.54 centímetros, o 0.0254 metros, y que cada henry tiene  $10^6$  microhenries. Exhíbanse todos los resultados con precisión de una cifra decimal. (Sugerencia: no se olvide hacer la conversión de pulgadas y vueltas por pulgadas a metros y vueltas por metro.)

Ejemplo de entrada:

100.0 5.0 1.0

Ejemplo de salida:

Dimensiones del solenoide:

Radio: 1.0 pulgadas  
 Longitud: 5.0 pulgadas  
 Vueltas/pulgadas: 100.0

Características eléctricas:

Inductancia: 5013.8 microhenries

- 6 La función  $\exp(x)$  de Pascal calcula un valor igual a la suma de la serie infinita

$$1 + x/1! + x^2/2! + x^3/3! + \dots$$

donde  $1! = 1$ ,  $2! = 2 * 1$ ,  $3! = 3 * 2 * 1$  y así sucesivamente. Supóngase que los datos de entrada contienen un solo valor real para  $x$  entre 0.0 y 1.0. Determínese la suma de los primeros cinco términos de la serie infinita y el valor de  $\exp(x)$  mediante la función estándar. Exhíbanse estos valores y el valor de  $x$  introducido acompañados de etiquetas apropiadas.

Ejemplo de entrada:

0.5

Ejemplo de salida:

Valor introducido = 5.000000000e-01

### ★ ★ PROBLEMAS IMPORTANTES

- 7 Con el resultado del ejercicio 12 del capítulo 2, esribase un programa para determinar la raíz cuadrada de un número positivo  $a$  mediante el cálculo de  $a^{0.5}$ . La entrada consistirá en una sola línea que contenga al número real  $a$ . Exibase  $a$ ,  $a^{0.5}$  y  $\text{sqrt}(a)$  con etiquetas apropiadas.

Ejemplo de entrada:

12.7

Ejemplo de salida:

|                         |                      |
|-------------------------|----------------------|
| Valor introducido       | = 1.2700000000e + 00 |
| Raíz cuadrada calculada | = 3.5637059362e + 00 |
| Raíz cuadrada de Pascal | = 3.5637059362e + 00 |

- 8 La entrada consiste en una sola línea que contiene un número real  $r$  y un entero  $p$ . Tómese nota de que  $p$  indica la posición del dígito al que se debe redondear  $r$ , de esta manera:

$$\begin{array}{r} r = x \ x \ x \cdot x \ x \ x \\ \uparrow \uparrow \uparrow \quad \uparrow \uparrow \uparrow \uparrow \\ p = 2 \ 1 \ 0 \ -1 \ -2 \ -3 \end{array}$$

Por ejemplo, si  $r = 35.89$  y  $p = -1$ , el valor redondeado deberá ser 35.9. Escribase un programa en Pascal que lleve a cabo esta operación de redondeo. (Sugerencia: piénsese en multiplicar por una potencia de 10, redondear mediante la función estándar *round* y dividir entre una potencia de 10.) Exibánse los valores introducidos ( $r$  y  $p$ ) y el valor redondeado. Utilicense etiquetas apropiadas para estos valores.

Ejemplo de entrada:

35.89 -1

Ejemplo de salida:

|                      |                           |
|----------------------|---------------------------|
| Valor introducido    | = 3.589000000000000e + 01 |
| Posición de redondeo | = -1                      |
| Valor redondeado     | = 3.590000000000000e + 01 |

- 9 Escribase un programa que tenga como única entrada un número octal (base ocho) de cinco dígitos. Exibase el número octal original y el número equiva-

lente en base 10. Por ejemplo, el número octal de tres dígitos 415 equivalente al valor decimal  $4 * 8^2 + 1 * 8^1 + 5 * 8^0$ , o sea 269. (Sugerencia: recuérdese que Pascal considerará el número introducido como número decimal.)

Ejemplo de entrada

217

Ejemplo de salida:

Octal 217 = Decimal 143

- 10 Dada una fecha expresada como cinco enteros:  $M$  (mes),  $D$  (día),  $S$  (siglo),  $A$  (año) y  $B$  (indicador de año bisiesto), es posible determinar el día de la semana correspondiente en forma de un entero en la escala de cero a seis. Este cálculo se puede realizar mediante la siguiente fórmula:

$$W = [D + (2.6 * M - 0.2) + A + (A/4) + (S/4) \\ - 2 * S - (1 + B) * (M/11)] \bmod 7$$

donde  $W$  = día de la semana (0 = domingo, 1 = lunes, y así sucesivamente)

$M$  = número del mes, *comenzando con marzo* = 1

$S$  = siglo (los dos primeros dígitos del año actual)

$A$  = año dentro del siglo (los dos últimos dígitos)

$D$  = día del mes

$B$  = 1 si el año es bisiesto, o en caso contrario

Las expresiones en paréntesis ( $M/11$ ), por ejemplo, representan el valor de las expresiones truncado a un entero. Considérese el primero de enero de 1985 como ejemplo. Se tiene  $M = 11$  (ya que marzo = 1),  $D = 1$ ,  $S = 19$ ,  $A = 85$  y  $B = 0$ . Por tanto,

$$W = [1 + (2.6 * 11 - 0.2) + 85 + (85 / 4) + (19 / 4) \\ - 2 * 19 - (1 + 0) * (11 / 11)] \bmod 7 \\ = (1 + 28 + 85 + 21 + 4 - 38 - 1 * 1) \bmod 7 \\ = 100 \bmod 7 \\ = 2.$$

Puesto que  $W = 2$ , se sabe que el primero de enero de 1985 cayó en martes. Los datos de entrada serán el número del mes actual (*con enero* = 1), el día del mes, el año (sin separar el *siglo* y el año dentro del siglo, como requiere la fórmula) y el indicador de año bisiesto (cero o uno). Determinese el día de la semana en que cayó la fecha que se introduce como datos. La salida debe incluir los datos de entrada y el entero que represente al día de la semana, todos con etiquetas apropiadas.

Ejemplo de entrada

3 12 1985 (que representa el 12 de marzo de 1985)

Mes = 3  
 Día = 12  
 Año = 1985  
 Día de la semana = 2

### ★★★PROBLEMA ESTIMULANTE

**N** Dado el importe real de un préstamo  $P$ , una tasa de interés real  $T$  y un número entero de años en los que se desea pagar el préstamo  $N$ , determíñese la cantidad  $C$  que se debe pagar cada año. Se puede utilizar la siguiente expresión para calcular  $C$ :

$$C = P * (1 + T/100)^N * T / 100 [(1 + T/100)^N - 1].$$

Los datos de entrada incluirán a  $P$ ,  $T$  y  $N$ . Calcúlese  $C$  e imprimánselo  $P$ ,  $T$ ,  $N$  y  $C$  con etiquetas y formatos apropiados. Supóngase que  $T$  se especifica como un porcentaje con un solo dígito fraccionario.

Ejemplo de entrada:

40000 10.0 30            (\$40,000 al 10% de interés durante 30 años)

Ejemplo de salida:

|                      |                   |
|----------------------|-------------------|
| Importe del préstamo | = \$40000.00      |
| Tasa de interés      | = 10.0 por ciento |
| Plazo del préstamo   | = 30 años         |
| Pago anual           | = \$4243.17       |



# CAPÍTULO 4

CAPÍTULO 4  
DISEÑO DESCENDENTE  
Y PROCEDIMIENTOS  
ELEMENTALES

Procedimientos  
y  
diseño  
descendente

Resolución  
de problemas  
mediante  
procedimientos  
simple

Procedimientos  
con  
parámetros

Resolución  
de problemas  
y  
procedimientos  
con  
parámetros

Diseño  
y prueba  
descendentes

**DISEÑO DESCENDENTE  
Y PROCEDIMIENTOS  
ELEMENTALES**

## OBJETIVOS

Después de completar este capítulo, el lector deberá ser capaz de:

- Aplicar el diseño descendente para resolver un problema dado
- Escribir un programa de computadora en términos de procedimientos
- Escribir un procedimiento con parámetros
- Distinguir entre parámetros formales y verdaderos
- Distinguir entre parámetros de entrada (de valor) y de salida (variables)
- Distinguir entre variables locales y globales
- Aplicar técnicas descendentes de prueba y depuración a los procedimientos

## PANORAMA GENERAL DEL CAPÍTULO

En este capítulo se analizarán aplicaciones de la estrategia de diseño descendente para la resolución de problemas. En particular, se hablará del proceso de dividir un problema en subproblemas y después refinar cada subproblema hasta hacer posible la resolución en computadora. La solución en computadora de cada subproblema se da en términos de subprogramas llamados **procedimientos**.

En la primera sección se analiza la colocación, invocación y ejecución de procedimientos simples, es decir, procedimientos sin parámetros. A continuación se presentará la solución detallada de un problema mediante procedimientos simples solamente. Se hará hincapié en la resolución de problemas mediante diseño descendente, desarrollo de algoritmos y resolución en computadora en términos de procedimientos. En la sección 3 se presenta la transferencia de información entre un procedimiento y otras partes del programa. Se analizan los parámetros formales y verdaderos, así como los de entrada (de valor) y de salida (variables). Se incluyen varios ejemplos para ilustrar la aplicación de procedimientos con parámetros. También se habla de las variables locales y globales.

El capítulo termina con una solución detallada de un problema más avanzado mediante procedimientos con parámetros. Además, se incluye una explicación sobre pruebas de procedimientos.

### SECCIÓN 4.1 INTRODUCCIÓN A LOS PROCEDIMIENTOS Y EL DISEÑO DESCENDENTE

Una estrategia para resolver un problema complejo consiste en dividirlo en subproblemas cuya resolución sea más sencilla. Estos subproblemas se pueden dividir a su vez en subproblemas más pequeños, y así sucesivamente hasta que sea fácil resolver los subproblemas más pequeños. A esta técnica de dividir el problema principal en subproblemas se le conoce en ocasiones como *divide y vencerás*. Durante este proceso se resuelve cada uno de los subproblemas. La estrategia de diseñar la solución de un problema principal mediante la resolución de sus

subproblemas se llama **diseño descendente**. Recibe este nombre porque se parte de “arriba” con un problema general y se diseñan soluciones específicas para sus subproblemas. Para obtener una solución efectiva del problema principal, es conveniente que los subproblemas sean independientes unos de otros. Así será posible resolver y probar cada subproblema por separado. Esta técnica es muy útil y permite manejar en forma efectiva la solución del problema principal complejo.

La conversión de las soluciones de problemas complejos obtenidas mediante el diseño descendente en soluciones por computadora se puede hacer fácilmente si se emplean lenguajes de alto nivel estructurados, como el Pascal. El problema principal se resuelve mediante el **programa principal** correspondiente (también llamado **impulsor**) del programa en Pascal. La solución de los subproblemas proviene de los subprogramas, llamados **procedimientos** o **funciones** en Pascal. Un procedimiento ejecuta las instrucciones de computadora que se requieren para resolver un subproblema dado. Si los programas de computadora se escriben en términos de procedimientos, es más fácil manejar la solución en computadora de un problema dado. En este capítulo se examinará con mayor detalle la correspondencia entre el diseño descendente de un problema y la solución en computadora en términos del programa principal y sus procedimientos (véase la Fig. 4-1).

Considérese una vez más el problema de calcular el área de un triángulo dado. Este problema se puede dividir en tres subproblemas:

**SUBPROBLEMA 1:** Introducir altura y base.

**SUBPROBLEMA 2:** Calcular el área.

**SUBPROBLEMA 3:** Exhibir los resultados.

El algoritmo escrito en seudocódigo quedaría entonces así:

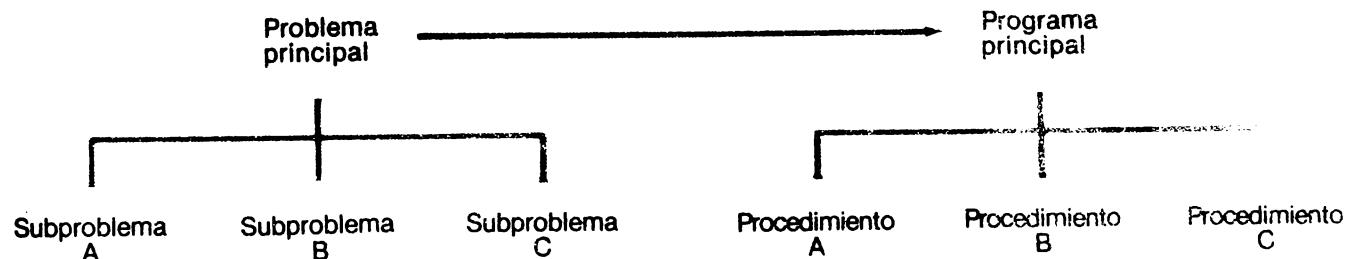
**PASO 1** Introducir los datos (altura y base).

**PASO 2** Calcular el área ( $\text{área} = 0.5 * \text{altura} * \text{base}$ ).

**PASO 3** Exhibir los resultados (altura, base, área).

A continuación se escribirá un procedimiento en Pascal para resolver cada subproblema.

**Figura 4-1** Diseño descendente y procedimientos.



- 1 El siguiente procedimiento en Pascal obtiene los datos de entrada:

```
PROCEDURE leedatos;
(*Obtener altura y base del triángulo *)
BEGIN
    write ('Introduzca la altura del triángulo:');
    readln (altura);
    write ('Introduzca la base del triángulo:');
    readln (base)
END;
```

Obsérvese que los procedimientos deben comenzar con la palabra reservada PROCEDURE (procedimiento) seguida de un identificador. Este identificador le da nombre al procedimiento (el que se acaba de escribir se llama *leedatos*). Obsérvese también que los procedimientos siempre llevan un signo de punto y coma después de la palabra END, ya que los procedimientos siempre van seguidos de algo más. Como siempre, no se requiere el punto y coma antes del END.

Fuera de los dos puntos que se mencionaron, la estructura de los procedimientos es muy similar a la de un programa completo en Pascal. De hecho, se verá más adelante que los procedimientos pueden incluir declaraciones de constantes y variables, así como información referente a los datos que va a leer y producir el procedimiento. Esto permite escribir los procedimientos en forma de módulos independientes del programa y autosuficientes.

- 2 El siguiente procedimiento calcula el área del triángulo, dadas la altura y la base:

```
PROCEDURE calculárea;
(* Calcular el área del triángulo a partir de base y altura *)
BEGIN
```

```
área := 0.5 * altura * base
END;
```

- 3 Por último, este procedimiento exhibe los resultados:

```
PROCEDURE darresultados;
(* Exhibir los resultados del cálculo del área de un triángulo *)
BEGIN
    writeln ('La altura del triángulo es', altura:6:2);
    writeln ('La base del triángulo es', base:6:2);
    writeln ('El área del triángulo es', área:6:2)
END;
```

Ahora ya se tienen soluciones en computadora de los tres subproblemas. En seguida se examinará la estructura del programa principal en el que se incorporarán estos procedimientos. Como se muestra a continuación, el programa principal contiene solamente tres proposiciones:

|               |                                                 |
|---------------|-------------------------------------------------|
| leedatos;     | (* invoca el proced. para obtener datos *)      |
| calculárea;   | (* invoca el proced. para calcular área *)      |
| darresultados | (* invoca el proced. para exhibir resultados *) |

END.

Obsérvese que las proposiciones que se emplearon en el programa principal consisten únicamente en los nombres de los procedimientos. Cuando se ejecuta en Pascal una proposición que se compone de un nombre de procedimiento, se “llama” o “invoca” al procedimiento, lo que ocasiona la ejecución de sus proposiciones. Después de ejecutar el procedimiento, el programa continúa en la siguiente proposición (la línea después de la invocación del procedimiento). Así, en el programa principal, la proposición

**leedatos;**

ordena a la computadora que invoque al procedimiento *leedatos* que resuelve el subproblema 1. Después de ejecutar el procedimiento *leedatos*, la computadora ejecutará la siguiente instrucción,

**calculárea;**

que, por supuesto, hace que el procedimiento *calculárea* determine el área del triángulo. Después de terminar el cálculo, la última proposición,

**darresultados**

invoca al procedimiento *darresultados* que se encarga de exhibir los resultados pertinentes.

El paso final es escribir el programa completo. Los únicos componentes que faltan son el encabezado del programa y las declaraciones. Además de la declaración de todas las variables, el Pascal requiere la declaración de todos los procedimientos. Estas declaraciones se colocan antes del programa principal y después de las declaraciones de variables. El programa completo en Pascal, cuyo nombre es *triángulo*, se muestra en seguida.

**PROGRAM** triángulo (input, output);

(\* Programa para calcular el área de un triángulo \*)

(\* dadas la base y la altura. \*)

**VAR**

|                     |                        |
|---------------------|------------------------|
| altura, base: real; | (* datos de entrada *) |
| área: real;         | (* área calculada *)   |

**PROCEDURE** leedatos;

```
(* Obtener altura y base del triángulo *)
BEGIN
    write ('Introduzca la altura del triángulo: ');
    readln (altura);
    write ('Introduzca la base del triángulo: ');
    readln (base)
END;
PROCEDURE calculárea;
(* Calcular el área del triángulo a partir de base y altura *)
BEGIN
    área := 0.5 * altura * base
END;

PROCEDURE darresultados;
(* Exhibir los resultados del cálculo del área de un triángulo *)
BEGIN
    writeln ('La altura del triángulo es', altura:6:2);
    writeln ('La base del triángulo es', base:6:2);
    writeln ('El área del triángulo es', área:6:2)
END;

(* El programa principal comienza aquí *)
BEGIN
    leedatos;      (* invoca el proced. para obtener datos *)
    calculárea;   (* invoca el proced, para calcular área *)
    darresultados  (* invoca el proced. para exhibir resultados *)
END.
```

---

### *Programa triángulo*

Ahora se ejecutará este programa a mano para los valores altura = 4.0 y base = 5.0. La primera proposición, *leedatos*, obtendrá los valores altura = 4.0 y base = 5.0. La siguiente proposición, *calculárea*, calculará el área del triángulo:

$$\text{área} = 5.0 * 4.0 * 5.0 = 10.0$$

La última proposición, *dar resultados*, exhibirá los resultados:

La altura del triángulo es 5.00  
 La base del triángulo es 4.00  
 El área del triángulo es 10.00

Se presentó un ejemplo sencillo para hacer hincapié en la correspondencia entre el diseño descendente del problema y la solución real en la computadora en términos de procedimientos. En muchos casos no se gana nada al escribir una secuencia de proposiciones en Pascal como procedimiento. Después de adquirir cierta experiencia en la resolución de problemas con diseño descendente, quedará

## Estructura de los procedimientos

Como ya se vio, los procedimientos en Pascal tienen una estructura que imita a la de un programa en Pascal. Tiene un encabezado, declaraciones de identificadores que se utilizan exclusivamente dentro de ese procedimiento (llamados identificadores *locales*), y por último las proposiciones ejecutables que se deben ejecutar cuando se invoque ese procedimiento.

El encabezado del procedimiento es el mecanismo que se emplea para dar nombre al procedimiento, del mismo modo que se usa el encabezado del programa para dar nombre al programa completo. En la siguiente sección se verá que el encabezado del procedimiento también sirve para identificar los valores que se pueden comunicar entre el procedimiento y su invocación. Examíñese este ejemplo de encabezado de procedimiento:

**PROCEDURE leedatos;**

La palabra reservada PROCEDURE (procedimiento) va seguida del identificador (*leedatos*) que se eligió para dar nombre al procedimiento. Obsérvese que el signo de punto y coma que sigue al identificador sirve en realidad para separar al encabezado del procedimiento de las proposiciones que le deben seguir (y que no se muestran arriba).

Es necesario utilizar solamente identificadores únicos para dar nombre a los procedimientos. La razón es obvia: si dos procedimientos (diferentes) se llamaran ambos *leedatos*, ¿cuál se usaría cuando se invocara a *leedatos*? Si sólo hay un procedimiento llamado *leedatos*, la selección no presentará ambigüedad, pero si hay más de uno, la computadora se verá obligada a tomar una decisión (y todavía no existen computadoras que piensen).

Después del encabezado del procedimiento se pueden incluir declaraciones (p. ej., declaraciones de constantes y de variables) de aquellos identificadores que se utilizarán exclusivamente en el procedimiento que se va a definir y que serán sus variables locales. Por ahora se omitirán estas declaraciones y se usarán las variables declaradas en el programa principal, que se llaman comúnmente variables *globales*. No obstantes, se volverá al tema de identificadores locales en la siguiente sección, cuando se analice la forma de hacer procedimientos que sean módulos independientes y autosuficientes.

El tercer componente de un procedimiento, las proposiciones ejecutables, siguen a las declaraciones o, en ausencia de éstas, al encabezado del procedimiento. Estas proposiciones van precedidas de la palabra reservada BEGIN y seguidas de la palabra reservada END, al igual que las del programa principal. Una vez más, un detalle que hay que recordar es que el END del programa principal va seguido de un punto para indicar que el programa ha terminado, mientras que el END de un procedimiento va seguido siempre de un punto y coma, ya que es indispensable que al procedimiento le siga otra porción de código del programa.

He aquí un resumen de las principales reglas para la escritura de procedimientos simples:

### *Resumen de procedimientos simples*

#### *Declaración*

- 1 El encabezado:
- 2 Las declaraciones:
- 3 Las proposiciones ejecutables:

```
PROCEDURE identificador;
(No se emplean en procedimientos simples)
BEGIN
    proposición;
    .
    .
    .
    proposición
END;
```

#### *Colocación*

Las declaraciones de procedimientos simples aparecen justo antes del BEGIN que inicia al programa principal y después de todas las declaraciones de constantes y variables.

#### *Invocación*

Para invocar un procedimiento simple se escribe su nombre como proposición ejecutable, seguida de un signo de punto y coma (si es necesario) para separarla de la siguiente proposición. Cuando se invoca un procedimiento, 1) la computadora "recuerda" la localización de la proposición que sigue a la invocación del procedimiento, 2) se ejecutan las proposiciones ejecutables del procedimiento y 3) la ejecución continúa en el programa que "llamó" al procedimiento en el punto que se recuerda.

## **Invocación y ejecución de procedimientos**

Puesto que los procedimientos son similares a los programas, normalmente es muy sencillo convertir un programa en un procedimiento. Como ejemplo de esto se muestra en seguida el programa *impuesto* de la sección 2.2 escrito de manera que utiliza un procedimiento llamado *unimpuesto*.

---

```
PROGRAM impuesto (input, output);
(* Programa que calcula impuesto de venta del 7% sobre artículos *)
(* adquiridos *)
CONST
    tasaimp = 0.07;
VAR
    costoart, impventa : real;
PROCEDURE unimpuesto;
BEGIN
    (* Obtener costo del artículo de datos de entrada *)
```

```

write ('Por favor escriba el costo del artículo: ');
readln (costoart);
(* Calcular impuesto por venta del artículo *)
impventa := tasaimp * costoart;
(* Exhibir resultados *)
writeln ('Costo del artículo      ', costoart:6:2);
writeln ('Impuesto por venta     ', impventa:6:2)
END;

BEGIN (* El programa principal *)
  unimpuesto (* invocar el procedimiento *)
END.

```

---

*Programa impuesto con el procedimiento unimpuesto*

El programa principal consta únicamente de una proposición:

```

BEGIN
  unimpuesto; (* invocar el procedimiento *)
END.

```

Esta proposición sirve para invocar al procedimiento *unimpuesto*. Probablemente será obvio que el escribir *unimpuesto* como procedimiento e invocarlo de esta forma no produce un beneficio inmediato. Si fuera necesario calcular el impuesto de varios artículos, entonces se podría invocar a *unimpuesto* varias veces, empleando una sola proposición cada vez. Resulta claro que esto es más eficiente que escribir varias veces todo el grupo de proposiciones contenidas en el procedimiento *unimpuesto*. Por ejemplo, supóngase que se desea calcular el impuesto de tres artículos. El programa principal sería:

```

BEGIN
  unimpuesto;      (* primera invocación *)
  unimpuesto;      (* segunda invocación *)
  unimpuesto       (* tercera invocación *)
END.

```

Se puede aprender mucho si se sigue la secuencia de pasos que ejecuta en realidad esta secuencia de proposiciones. Como auxiliar, imagínese un apuntador (un “dedo”) que indica la siguiente proposición que va a ejecutar la computadora. Si el programa principal incluyera únicamente las tres proposiciones que se muestran arriba, el apuntador estaría colocado en la primera de las tres cuando comenzara el programa. Esto se ilustrará así:

→ unimpuesto;  
 unimpuesto;  
 unimpuesto

Cuando se ejecuta esta proposición (una invocación de procedimiento), suceden dos cosas importantes, *en el orden que se da a continuación*.

- 1 Se toma nota de la posición de la proposición después de la invocación de procedimiento y se guarda para hacer referencia a ella más tarde. (Esta posición se indicará con un apuntador así:  $\Rightarrow$ ).
- 2 El apuntador de dedo (el que señala la siguiente proposición que se va a ejecutar) se mueve a la primera proposición ejecutable del procedimiento invocado. (Cuando se incluyen parámetros, la invocación es más compleja, pero esto se estudiará más adelante.)

Ahora se seguirán los pasos de la invocación y ejecución del procedimiento

### Paso 1.

Comienza la ejecución del programa.

| <i>programa principal</i>                              | <i>procedimiento</i>                                                                                                                                                                                                                                                                                                                     |
|--------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\rightarrow$ unimpuesto;<br>unimpuesto;<br>unimpuesto | <pre>PROCEDURE unimpuesto; BEGIN   writeln ('Por favor escriba el costo del artículo');   readln (costoart);   (* Calcular impuesto por venta del articulo *)   impventa := tasaimp * costoart;   (* Exhibir resultados *)   writeln ('Costo del artículo: ', costoart:6:2);   writeln ('Impuesto por venta: ', impventa:6:2) END;</pre> |

### Paso 2.

Se invoca al procedimiento *unimpuesto*. Obsérvese la posición del apuntador de regreso ( $\Rightarrow$ ) y del apuntador de la siguiente proposición ( $\rightarrow$ ).

| <i>programa principal</i>                              | <i>procedimiento</i>                                                                                                                                                                                                                                                                                                                                               |
|--------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| unimpuesto;<br>$\Rightarrow$ unimpuesto;<br>unimpuesto | <pre>PROCEDURE unimpuesto; BEGIN   <math>\rightarrow</math> writeln ('Por favor escriba el costo del artículo:');   readln (costoart);   (* Calcular impuesto por venta del articulo *)   impventa := tasaimp * costoart;   (* Exhibir resultados *)   writeln ('Costo del artículo: ', costoart:6:2);   writeln ('Impuesto por venta: ', impventa:6:2) END;</pre> |

Se ejecuta la proposición “writeln (‘Por favor escriba el costo del artículo’)”.

| <i>programa principal</i>                  | <i>procedimiento</i>                                                                                                                                                                                                                                                                                                                                                      |
|--------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| unimpuesto;<br>→ unimpuesto;<br>unimpuesto | <pre> PROCEDURE unimpuesto; BEGIN     writeln ('Por favor escriba el costo del     artículo');     → readln (costoart);     (* Calcular impuesto por venta del     artículo *)     impventa := tasaimp * costoart;     (* Exhibir resultados *)     writeln ('Costo del artículo: ',     costoart:6:2);     writeln ('Impuesto por venta: ',     impventa:6:2) END;</pre> |

#### Pasos 4 a 7.

Se ejecutan las demás proposiciones de *unimpuesto*. (Recuérdese que se hace caso omiso de los comentarios durante la ejecución.)

| <i>programa principal</i>                  | <i>procedimiento</i>                                                                                                                                                                                                                                                                                                                                    |
|--------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| unimpuesto;<br>→ unimpuesto;<br>unimpuesto | <pre> PROCEDURE unimpuesto; BEGIN     writeln ('Por favor escriba el costo del artículo');     readln(costoart);     (* Calcular impuesto por venta del artículo *)     impventa := tasaimp * costoart;     (* Exhibir resultados *)     writeln('Costo del artículo: ', costoart:6:2);     writeln('Impuesto por venta: ', impventa:6:2); → END;</pre> |

#### Paso 8.

El procedimiento termina y la ejecución continúa después de la invocación. Tómese nota de que el apuntador de regreso se convierte en el apuntador de la siguiente proposición y que, además, no quedan apuntadores en el procedimiento.

| <i>programa principal</i>                               | <i>procedimiento</i>                                                                                                                                                                                                                                                                                                                                                 |
|---------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> unimpuesto; → unimpuesto;       unimpuesto </pre> | <pre> PROCEDURE unimpuesto; BEGIN       writeln('Por favor escriba el costo del artículo');       → readln(costoart);       (* Calcular impuesto por venta del artículo *)       impventa := tasaimp * costoart;       (* Exhibir resultados *)       writeln('Costo del artículo: ', costoart:6:2);       writeln('Impuesto por venta: ', impventa:6:2) END; </pre> |

Los pasos 2 al 8 se repiten para cada una de las dos invocaciones del procedimiento restantes, exactamente como antes (con la salvedad de la posición del apuntador de regreso). Naturalmente, existe el potencial para procesar datos diferentes cada vez. Es en esta capacidad de realizar las mismas acciones con diferentes grupos de datos donde reside el poder real de los procedimientos.

Ahora se probará la ejecución a mano de este programa con números reales. Supóngase que los tres artículos cuestan \$100, \$200 y \$300, respectivamente. Estos números serán los datos de entrada para el problema. Después de la primera invocación del procedimiento *unimpuesto*, se exhibirán los siguientes renglones:

|                     |        |
|---------------------|--------|
| Costo del artículo: | 100.00 |
| Impuesto por venta: | 7.00   |

Después de la segunda invocación se exhibirá lo siguiente:

|                     |        |
|---------------------|--------|
| Costo de artículo:  | 200.00 |
| Impuesto por venta: | 14.00  |

Por último, esto es lo que se exhibirá después de la invocación final:

|                     |        |
|---------------------|--------|
| Costo del artículo: | 300.00 |
| Impuesto por venta: | 21.00  |

### Cuándo conviene emplear procedimientos

No en todos los programas son necesarios los procedimientos. De hecho, puede haber muchos programas útiles de cientos de líneas que jamás utilicen un procedimiento. Entonces, ¿cuándo debe usarse un procedimiento? ¿Qué criterios se deben aplicar para tomar la decisión?

Una de las razones más obvias para escribir una secuencia de proposiciones como un procedimiento es que la secuencia se utiliza varias veces en diferentes puntos del programa. Sin duda, el programa seguirá siendo correcto si se escriben esas proposiciones en cada punto del programa donde se necesiten. Sin embargo, esta estrategia presenta varios problemas.

En primer lugar y obviamente, el programa será más grande de lo necesario. Esto dificultará la lectura, escritura, captura y mantenimiento (hacer modificaciones y corregir errores) del programa. Además, los programas fuente grandes requieren una porción mayor de la memoria de la computadora (lo cual es una consideración muy importante para los usuarios de computadoras pequeñas).

En segundo lugar, se verá con menor claridad la naturaleza del cálculo que realiza el programa. Es de esperar que un cálculo determinado se lleve a cabo de la misma manera en todos los puntos del programa en que se necesita. Si esto no se hace así, el lector del programa abrigará sospechas y se preguntará, “¿por qué se repite este código? ¿Existe alguna diferencia sutil entre ésta y su anterior aparición?”

Lo anterior se aplica al uso de procedimientos motivado por repeticiones en el código. ¿Existe otra razón para incluir procedimientos en los programas? Definitivamente. Y no tiene nada que ver con código repetido. La agrupación de proposiciones, como la que se hace cuando se escribe un procedimiento, sugiere que esas proposiciones tienen un objetivo particular. En seguida se muestra un ejemplo sencillo de lo anterior:

```
BEGIN          (* programa principal *)
    leedatos;
    procesadatos;
    darresultados
END.
```

Todas las líneas entre BEGIN y END son invocaciones de procedimientos. Los nombres de los procedimientos se eligieron de manera que sugirieran las funciones que realizan los procedimientos: leer ciertos datos, procesarlos de alguna manera y después imprimir determinados resultados. No se sabe específicamente cuál es el procesamiento que va a realizar el programa, pero sí se sabe mucho acerca de su estructura. Se sabe que las proposiciones del procedimiento *leedatos* se encargarán de todos los detalles de la lectura de datos de entrada, que el procedimiento *procesadatos* incluirá las proposiciones que constituyen la parte medular del cómputo y que el procedimiento *darresultados* se encargará de la exhibición de los resultados. No cabe duda que esta parte del código es muy legible, aun sin comentarios.

Ahora bien, existe la posibilidad de que cada uno de los procedimientos del mismo programa sea escrito por diferentes programadores. Siempre que el problema se haya subdividido cuidadosamente y se haya reducido al mínimo la interacción entre las soluciones individuales, cada uno de los procedimientos puede ser totalmente independiente de los demás. Esta posibilidad de asignar a distintos programadores el desarrollo de procedimientos para el mismo programa es muy importante para la escritura de programas grandes (que pueden incluir decenas o hasta cientos de procedimientos).

Al lector puede parecerle que se insiste demasiado en la legibilidad de los programas. No obstante, conviene subrayar en este punto que la mayor parte del tiempo que se dedica a los programas no se invierte en su invención, sino en su mantenimiento. Es decir, las tareas asociadas a la corrección y modificación del programa consumen más tiempo de programador que su creación original. Es de

esperarse que la descomposición correcta de programas en procedimientos mejore esta situación.

## Colocación de los procedimientos y tipos rígidos

Como se aprendió en el capítulo 2, es indispensable declarar (en un encabezado, declaración CONST o declaración VAR) todos los identificadores que se emplean en un programa en Pascal antes de poderlos usar. Esta regla se aplica también a los procedimientos. Para *declarar* un procedimiento, es necesario escribir el procedimiento completo (con encabezado, declaraciones locales y proposiciones ejecutables) antes de las proposiciones ejecutables del programa que lo utiliza. De modo específico los procedimientos se colocan inmediatamente antes de la palabra reservada BEGIN que marca el comienzo de las proposiciones ejecutables del programa principal. (Más adelante se verá que los procedimientos también pueden aparecer *dentro* de otros procedimientos.)

Es posible que al principio parezca un poco extraña esta colocación de los procedimientos *antes* de las proposiciones que los utilizan. Después de todo, las proposiciones ejecutables del procedimiento no se van a ejecutar sino hasta después de que una proposición posterior los invoque. Entonces, ¿por qué es preciso colocarlos antes?

Para comprender la respuesta a esta pregunta hay que entender primero las reglas de *tipos rígidos* de Pascal. El Pascal limita el uso de identificadores a aquellos contextos en los que el identificador tiene sentido. Por ejemplo, no tiene sentido tratar de sumar cuatro al procedimiento *unimpuesto* que se presentó antes. De manera similar, es absurdo sumar tres a una variable booleana. El compilador de Pascal detectará todos los intentos de realizar operaciones absurdas como éstas *durante la traducción del programa*. En algunos otros lenguajes de programación, estos errores no se detectan sino hasta después (durante la ejecución del programa) y es posible que para entonces ya hayan causado problemas serios.

Entonces, ¿por qué exigen las reglas de tipos rígidos que se declaren los procedimientos *antes* de invocarlos? La respuesta es que el compilador de Pascal debe saber, antes de la invocación, que un identificador se refiere a un procedimiento para poder hacer la verificación de tipo necesaria.

## SECCIÓN 4.2 RESOLUCIÓN DE PROBLEMAS MEDIANTE PROCEDIMIENTOS SIMPLES

En esta sección se presenta la solución detallada de un problema por medio de procedimientos simples.

### Problema 4.1

*Dado un reloj de 24 horas y un tiempo representado en forma de entero de la forma hhmm, donde hh representa las horas (00 a 23) y mm representa los minutos (00 a 59), determíñese el tiempo después de que han pasado cinco horas*

y 23 minutos. Los resultados deberán expresarse en la misma forma que los datos de entrada. Por ejemplo, si el dato de entrada es 1230 (que representa las 12:30 P.M.), la salida deberá ser 1753 (que representa las 5:53 P.M.).

Este problema se puede dividir en tres subproblemas:

SUBPROBLEMA 1: Introducir el tiempo.

SUBPROBLEMA 2: Calcular el nuevo tiempo.

SUBPROBLEMA 3: Exhibir el nuevo tiempo.

Los subproblemas 1 y 3 se pueden resolver mediante proposiciones de entrada/salida sencillas. El subproblema 2 requiere un análisis más profundo. Supóngase que el dato de entrada es 1230 (entero). En este caso, para calcular el tiempo 5 horas y 23 minutos después, bastaría con sumar directamente 0523 a 1230 para producir el resultado correcto, 1753. Pero el programa debe funcionar correctamente para cualquier tiempo que se proporcione como dato de entrada. Si el dato fuera 2359 (un minuto antes de la media noche), es obvio que la respuesta debe ser 0522 (que representa las 5:22 A.M.). Sin embargo, la suma directa de 2359 y 0523 da 2882, lo que obviamente es erróneo.

El problema se debe a la forma en que se presentan los datos de entrada. Se proporciona un solo entero en una sola "unidad" que representa a dos enteros en dos diferentes unidades (horas y minutos). Las salidas deben tener la misma forma, por lo que es evidente que el subproblema 2 se puede subdividir en tres subproblemas:

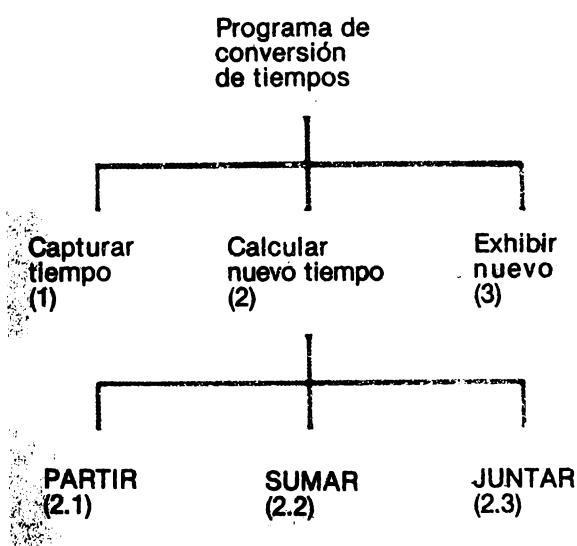
SUBPROBLEMA 2.1: (*Partir*) Separar el dato de entrada en horas y minutos.

SUBPROBLEMA 2.2: (*Sumar*) Sumar 5 a las horas y 23 a los minutos.

SUBPROBLEMA 2.3: (*Juntar*) Convertir horas y minutos en un entero para exhibirlo.

El diseño descendente general de este problema se muestra en la figura 4-2.

**Figura 4-2** Diseño descendente del problema de conversión de tiempos.



Para resolver subproblema 2.1, es preciso encontrar la forma de partir un entero como 1230 en dos enteros separados, 12 y 30. Se puede utilizar la división entera y la operación de residuo, representadas en Pascal por los operadores DIV y MOD, para lograr la separación. Esta técnica, que se examinará en seguida, se puede aplicar en muchos otros problemas parecidos.

Obsérvese que la división entera entre 100 elimina en forma efectiva los dos dígitos decimales del número que se divide. Por ejemplo, 1230 DIV 100 es igual a 12, habiéndose “eliminado” el 30. Adviértase también que un número MOD 100 da como resultados de hecho, los dos dígitos de la derecha. Así, 1230 MOD 100 produce 30, y se eliminan los dígitos de orden más alto.

Una vez separados los componentes de horas y minutos de los datos de entrada, se puede sumar 5 a las horas y 23 a los minutos para resolver el subproblema 2.2. Pero todavía no es posible pasar al subproblema 3, ya que esta suma puede ocasionar una cantidad de minutos mayor de 59 o una cantidad de horas mayor de 23. Por tanto, es posible que estos valores requieran un ajuste antes de la conversión a un solo entero. Se puede utilizar una vez más la operación de división (el operador DIV en Pascal) para determinar el número de horas completas en la cifra de minutos resultante y la operación de residuo (el operador MOD en Pascal) para extraer los minutos que sobran de esa cifra. En seguida, el subproblema 2.3 convertirá las horas y minutos en un solo entero, con el fin de exhibirlo. Para ello se multiplicarán las horas por 100 y se sumarán los minutos.

Considérense las horas y minutos que resultan del dato de entrada 2159. Después de sumar los minutos, la nueva cifra de minutos será 59 más 23, o sea 82. Esto representa una hora con 22 minutos, y se puede calcular en Pascal con 82 DIV 60 (una hora) y 82 MOD 60 (22 minutos). Después de sumar 5 a las horas originales, las horas que se extraigan de la cifra de minutos se deberán agregar a la nueva cifra de horas. Después, se deberá convertir este valor en un número en la escala de cero a 23, otra vez mediante el operador MOD. Con el dato de entrada 2159 se obtiene un valor de 21 para las horas. Al sumar 5 se tiene 26 y después de sumar el excedente de una hora obtenido de la cifra de minutos se tendrá un valor de 27 horas. Éste se convierte en 3 horas mediante la expresión 27 MOD 24. La siguiente tabla resume estos resultados.

#### SUBPROBLEMA 2.1

|             |                          |                          |
|-------------|--------------------------|--------------------------|
| entrada     | horas                    | minutos                  |
| <b>2159</b> | <b>2159 DIV 100 = 21</b> | <b>2159 MOD 100 = 59</b> |

#### SUBPROBLEMA 2.2

|                   |                               |
|-------------------|-------------------------------|
| Combinar minutos: | <b>59 + 23 = 82</b>           |
| Ajustar minutos:  | <b>82 DIV 60 = 1 hora</b>     |
|                   | <b>82 MOD 60 = 22 minutos</b> |

#### SUBPROBLEMA 2.3

|                    |                                           |
|--------------------|-------------------------------------------|
| Combinar horas:    | <b>21 + 5 = 26</b>                        |
| Incrementar horas: | <b>26 + 1 = 27 (si minutos &gt; = 60)</b> |
| Ajustar horas:     | <b>27 MOD 24 = 3 (si horas &gt; = 24)</b> |

*Seudocódigo del problema 4.1*

PASO 1 Capturar tiempo: obtener un dato de entrada que represente el tiempo en el reloj de 24 horas.

PASO 2 Calcular el nuevo tiempo.

PASO 2.1 Separar el dato de entrada en horas y minutos.

PASO 2.2 Sumar 5 horas y 23 minutos.

PASO 2.3 Convertir horas y minutos en entero apropiado para exhibirse.

PASO 3 Exhibir el valor que resulta del paso 2.3

En seguida se muestra el programa completo en Pascal (con procedimientos) para el problema, así como ejemplos de las salidas. Obsérvese que se usaron dos procedimientos, *partir* y *juntar*, para convertir la representación entera de un tiempo en la escala de 24 horas en las horas y minutos equivalentes, y para hacer la conversión de horas y minutos a un entero. Cabe hacer notar que no hubiera sido provechoso escribir las proposiciones de entrada y salida en forma de procedimientos. Sin embargo, *partir* y *juntar* son útiles y constituyen ejemplos importantes de la manera de *encapsular* varias tareas necesarias para la resolución de problemas. De hecho, se podrían emplear estos procedimientos en muchos otros problemas que manejan tiempos representados en un reloj de 24 horas. Los ejercicios, por ejemplo, incluyen el problema general de sumar dos tiempos arbitrarios para obtener un resultado. Tener ya escritos y probados los procedimientos *partir* y *juntar* representa un avance considerable en la resolución de ese problema.

---

PROGRAMA convertiempo (input, output);

CONST

hora = 60; (\* minutos por hora \*)  
día = 24; (\* horas por día \*)

VAR

tiempoe, (\* dato de entrada \*)  
tiempor, (\* tiempo resultante \*)  
mins, (\* minutos separados \*)  
hrs : integer; (\* horas separadas \*)

PROCEDURE partir;

(\* Procedimiento para separar un tiempo entero de formato "hhmm" \*)

(\* en la variable "tiempos" en horas y minutos separados en \*)

(\* las variables "hrs" y "mins". \*)

BEGIN

hrs := tiempoe DIV 100; (\* elimina los minutos \*)

mins := tiempoe MOD 100 (\* elimina las horas \*)

END;

```

PROCEDURE juntar;
(* Procedimiento para almacenar en la variable "tiempor" el valor *)
(* entero en la forma "hhmm" que corresponde a "hrs" horas y "mins" *)
(* minutos. Las horas que sobran en "mins" se suman a "hrs" y se *)
(* eliminan los días sobrantes en "hrs". *)
BEGIN
    hrs := hrs + mins DIV hora;      (* sumar horas que sobran *)
    mins := mins MOD hora;          (* eliminar horas que sobran *)
    hrs := hrs MOD día;             (* eliminar días que sobran *)
    tiempor := hrs * 100 + mins    (* calcular tiempo nuevo *)
END;

BEGIN (* programa principal *)
(* Pedir al usuario el dato de entrada *)
write ('Escriba un tiempo en la forma "hhmm": ');
readln (tiempoe);
(* Separar tiempoe en hrs y mins *)
partir;
(* Sumar cinco horas y 23 minutos *)
hrs := hrs + 5;
mins := mins + 23;
(* Ajustar el tiempo y crear el tiempo resultante *)
juntar;
(* Exhibir el tiempo resultante *)
writeln ('El tiempo 5:23 más tarde es ', tiempor)
END.

```

---

*Programa convertiempo*

Escriba un tiempo en la forma "hhmm": 1230  
 El tiempo 5:23 más tarde es 1753

Escriba un tiempo en la forma "hhmm": 2150  
 El tiempo 5:23 más tarde es 313

*Programa convertiempo: ejemplos de ejecución***EJERCICIOS DE LA SECCIÓN 4.2**

- 1 Especifíquese el resultado que se obtiene al ejecutar el siguiente programa

```

PROGRAM velocidad (input, output);
VAR distancia, vel, tiempo : integer;
PROCEDURE velresult;
```

```

BEGIN
    vel := distancia DIV tiempo;
    writeln ('La velocidad es ', vel, 'Kph.');
END;
(* El programa principal comienza aquí *)
BEGIN
    distancia := 100;
    tiempo := 2;
    velresult;
    distancia := 400;
    tiempo := 10;
    velresult
END.
  
```

- 2 Especifíquese la salida que se obtiene cuando se ejecuta el siguiente programa.

```

PROGRAMA ejercicio (input, output);
VAR a, b, c : integer;
    PROCEDURE cambioa;
    BEGIN
        a := a + 1
    END;
    PROCEDURE cambiob;
    BEGIN
        b := b + 2
    END;
    PROCEDURE cambioc;
    BEGIN
        c := c - 1
    END;
(* El programa principal comienza aquí *)
BEGIN
    a := 1;
    b := 2;
    c := 3;
    cambioa;
    cambiob;
    cambioc;
    writeln (a, b, c);
    cambioa;
    cambiob;
    writeln (a, b, c)
END.
  
```

- 3 Escribáse un procedimiento en Pascal llamado *trisuma* que calcule la suma de las tres variables *núm1*, *núm2* y *núm3* (supóngase que las variables se declaran como enteros en el programa principal) y almacene la suma en la variable *total*.

- 4 Escribase un procedimiento en Pascal llamado *media* que divida el valor de la variable entera *total* entre 3 (mediante división entera) y almacene el resultado en la variable entera *promedio*.
- 5 Escribase un programa completo en Pascal llamado *calcular* que obtenga tres valores enteros para las variables *núm1*, *núm2* y *núm3*, que en seguida invoque a los procedimientos *trisuma* y *media* de los ejercicios 3 y 4 y que, por último, exhiba el valor promedio con un mensaje apropiado. Declárense todas las variables en el programa principal.
- 6 Escribase un procedimiento llamado *círculo* que calcule el área de un círculo a partir del radio, el cual se declarará como real en el programa principal.
- 7 Una variable llamada *níqueles* representa un cierto número de monedas de cinco centavos de dólar. Describase la finalidad del siguiente procedimiento (suponiendo que las variables *níqueles* y *valor* se declaran en el programa principal).

```
PROCEDURE vale;
BEGIN
    VALOR := 0.05 * níqueles
END;
```

- 8 Las variables *níqueles*, *dieces* y *cuartos* representan el número de monedas de cinco, 10 y 25 centavos de dólar, respectivamente, que se encuentran en el cajón de una máquina registradora. Escribase un procedimiento llamado *dinero* que determine el número equivalente de dólares en la máquina registradora y almacene el resultado en la variable real *dólares*. Supóngase que todas las variables se declaran en el programa principal.

### SECCIÓN 4.3 PROCEDIMIENTOS CON PARÁMETROS

Los procedimientos más efectivos son los módulos de programa independientes y autosuficientes. La razón de ello es que cuando un problema es muy complicado, los programas que se escriban para resolverlo probablemente serán también muy complejos. Para obtener una solución efectiva del problema, se aplica un diseño descendente y se divide el problema en subproblemas. Si los procedimientos que resuelven los subproblemas son módulos autosuficientes, es posible resolver y probar cada procedimiento en forma *independiente* con respecto a los demás procedimientos. Casi siempre esto descompone un problema difícil en partes más manejables. Los gerentes de grandes proyectos de programación utilizan muchas veces la estrategia de divide y vencerás.

Los procedimientos simples que se usaron anteriormente no son autosuficientes porque operan sobre variables específicas que se declaran en el programa principal. Para que los procedimientos sean programas autosuficientes, muchos lenguajes de alto nivel permiten la *transferencia* de información entre los procedimientos y otras partes del programa mediante objetos llamados *parámetros*. Los parámetros permiten a los procedimientos manipular diferentes grupos de valores, de manera

que se puede utilizar el mismo procedimiento varias veces en el mismo programa. Antes de analizar el uso de parámetros con procedimientos en Pascal, se estudiará la similitud entre un procedimiento y un programa.

Un programa de computadora, al igual que una computadora digital, se puede considerar como un procesador de información con datos de entrada y salida. De manera similar, si un procedimiento se considera como un programa autosuficiente, también será un procesador de información con entradas y salidas (parámetros). Véase la figura 4-3.

La información y los datos se pueden proporcionar como entradas a los procedimientos y éstos producirán salidas durante su ejecución.

Se regresará ahora al problema del área de un triángulo. Supóngase que se desea escribir un procedimiento autosuficiente e independiente en Pascal para determinar el área de un triángulo. ¿Qué información necesitará este procedimiento? Ciertamente, bastará con tener la altura y la base como datos de entrada. La salida es el área del triángulo. En un procedimiento en Pascal esta información (llamada *parámetros*) se puede incluir en el encabezado del procedimiento. Así, en el caso del problema del área de un triángulo se tendrán los siguientes parámetros:

Parámetros de entrada: altura y base

Parámetros de salida: área

El procedimiento en Pascal correspondiente (*triángulo*) con la inclusión de parámetros queda como sigue:

---

```

PROCEDURE triángulo (altura, base : real; VAR área : real);
(* Procedimiento para calcular el área de un triángulo      *)
(* dadas la altura y la base.                                     *)
BEGIN
  área := 0.5 * altura * base
END;

```

---

#### *Procedimiento triángulo*

En Pascal, los parámetros que se mencionan en el encabezado del procedimiento se llaman *parámetros formales*. Funcionan como representantes de los va-

**Figura 4-3** Semejanzas entre programas y procedimientos.



lores verdaderos que se proporcionan cuando se invoca el procedimiento. En el procedimiento *triángulo*, ¿cuál es el valor de la altura? Puesto que altura es un parámetro formal, no tendrá valor hasta que el programa que llama invoque al procedimiento y le pase el valor verdadero de la altura.

Examíñese con atención el encabezado del procedimiento. Después del nombre del procedimiento se tiene una lista de los parámetros de entrada (llamados *parámetros de valor* en Pascal) *altura* y *base*, seguidos de un signo de dos puntos y el tipo de dato de los parámetros. Pascal exige que se especifiquen los tipos de datos de todos los parámetros en la lista de parámetros. La lista de parámetros de entrada termina con un signo de punto y coma. Después de esto viene la lista de parámetros de salida (llamados *parámetros variables* en Pascal). Esta lista comienza con la palabra reservada VAR seguida de los parámetros de salida y su tipo de dato. La figura 4-4 ilustra lo anterior.

Conviene hacer un resumen de lo dicho sobre la lista de parámetros de este procedimiento. Los parámetros de entrada (de valor) *altura* y *base* son del tipo real y el parámetro de salida (variable, o VAR, en Pascal) *área* también es del tipo real. En el capítulo 7 se examinarán con mayor detalle los parámetros de valor y variables. En este punto es importante recordar lo siguiente para poder escribir procedimientos independientes y autosuficientes:

- Toda la información o datos que necesite un procedimiento deberá aparecer en la lista de parámetros formales como parámetros de entrada (de valor), sin la palabra reservada VAR.
- Todos los resultados obtenidos por el procedimiento y que necesita el programa que llamó al procedimiento deben aparecer en la lista de parámetros formales como parámetros de salida (variables) y deben ir precedidos de la palabra reservada VAR.

Es importante darse cuenta de que los parámetros que se listan en el encabezado del procedimiento son parámetros formales. De hecho, cambiar los nombres de los parámetros no cambiará los efectos del procedimiento. Por ejemplo, examíñese la siguiente versión del procedimiento *triángulo*:

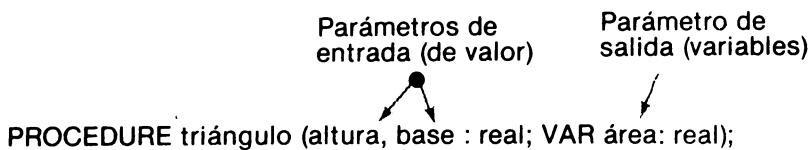
---

```
PROCEDURE triángulo (alt, tribase : real; VAR triárea : real);
(* Procedimiento para calcular el área de un triángulo *)
(* dadas la altura y la base. *)
BEGIN
    triárea := 0.5 * alt * tribase
END;
```

---

#### *Procedimiento triángulo, modificado*

**Figura 4-4** Encabezado de procedimiento con parámetros formales.



Esta versión del procedimiento es efectivamente la misma que la anterior. Se cambió el nombre de los parámetros de entrada *altura* y *base* a *alt* y *tribase*, y el parámetro de salida *área* se cambió a *triárea*. Los identificadores que se emplean para dar nombre a los parámetros formales solamente representan a los valores verdaderos, por lo que no dependen de los identificadores que se empleen en otros procedimientos y en el programa principal.

Más adelante (en el Cap. 7) se verá que se pueden usar parámetros VAR tanto para entrada como para salida. Se evitará este uso de los parámetros VAR mientras no se haya analizado el tema con mayor profundidad.

## Invocación de procedimientos con parámetros

Existe otro aspecto importante de los procedimientos con parámetros, que es la forma como se proporcionan y se recuperan los valores verdaderos de los parámetros formales. Se comenzará por volver una vez más al problema del área de un triángulo. Si se quisiera obtener el área de varios triángulos diferentes se utilizaría sin duda un procedimiento para realizar el cálculo del área. Por ejemplo, supóngase que se desea calcular el área de un triángulo con una altura de 3.0 unidades y una base de 4.0 unidades y almacenar el área en una variable llamada *triárea*. Esto puede hacerse mediante la siguiente invocación de procedimiento en el programa principal:

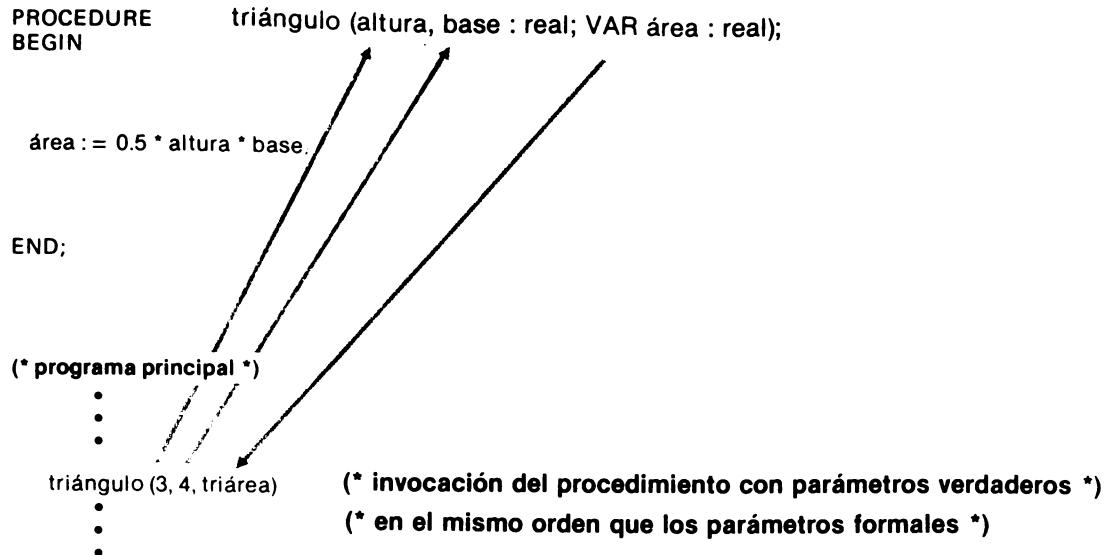
**triángulo (3.0, 4.0, triárea)**

La lista entre paréntesis que sigue al nombre del procedimiento se denomina *lista de parámetros verdaderos*. El orden de los parámetros verdaderos en la invocación del procedimiento debe ser el mismo que el orden de los parámetros formales en el encabezado del procedimiento.

La proposición de invocación del procedimiento se ejecuta en varios pasos. Para cada parámetro formal de entrada (de valor) se almacena una copia del valor del parámetro verdadero en una nueva localidad de memoria y se le da el nombre del parámetro formal correspondiente. En vez de crear una localidad de memoria nueva para los parámetros de salida (variables), se da a las localidades de memoria originales de cada uno un nombre adicional (un alias) conforme lo especifican los parámetros formales de salida (variables) correspondientes. La diferencia es importante, como se verá en breve.

En seguida se “recuerda” la localidad de la proposición que sigue a la invocación del procedimiento y se ejecuta la primera proposición ejecutable del procedimiento invocado. Estos últimos pasos son, como recordará el lector, exactamente los mismos que en el caso de los procedimientos simples (sin parámetros) que se explicaron anteriormente.

Ahora se aplicarán estos pasos a la invocación del primer procedimiento del área del triángulo. En primer lugar se almacenan los valores 3.0 y 4.0 en localidades de memoria nuevas llamadas *altura* y *base*, respectivamente. Después se asocia un nombre adicional, *área*, a la localidad de memoria que corresponde a la localidad de memoria de la variable llamada *triárea*. Este nombre nuevo es un alias de *triárea* que se conoce únicamente en el procedimiento *triángulo*. La correspon-



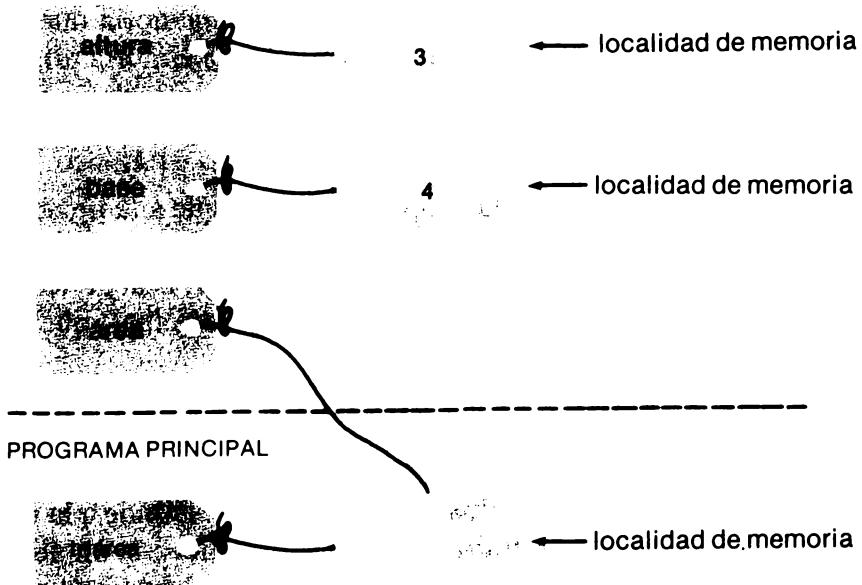
**Figura 4-5** Invocación de un procedimiento y transferencia de parámetros verdaderos.

dencia entre los parámetros verdaderos y formales se muestra en la figura 4-5. Obsérvese que se pasan al procedimiento los parámetros verdaderos, 3 y 4, y se almacenan en las localidades de memoria de los parámetros formales de entrada (de valor) *altura* y *base*. El parámetro formal de salida (variable) *área* y el parámetro verdadero *triárea* se refieren a la misma localidad de memoria que contiene el resultado que se devuelve al programa principal. La figura 4-6 ilustra estos conceptos.

El Pascal verifica (durante la compilación) que concuerden los tipos de los parámetros verdaderos y los parámetros formales correspondientes. Con un solo ti-

**Figura 4-6** Nombres y localización de los parámetros.

PROCEDURE triángulo



po de parámetro verdadero que no concuerde con el tipo del parámetro formal correspondiente, el compilador producirá un mensaje de error apropiado y no habrá programa ejecutable.

El parámetro verdadero que corresponda a un parámetro de entrada (de valor) puede ser una constante o una expresión, ya que se crea una copia de su valor cuando se invoca el procedimiento. Por ejemplo, podría emplearse el siguiente segmento de programa para calcular el área del mismo triángulo que en el ejemplo anterior (las variables *base*, *alt* y *triárea* se deben declarar como variables reales en el programa principal):

```
alt := 3.0;
base := 4.0;
triángulo (alt, base, triárea); (* invocación del procedimiento *)
```

O bien se podrían leer los valores de *alt* y *base* antes de invocar el procedimiento:

```
read (alt, base);
triángulo (alt, base, triárea); (* invocación del procedimiento *)'
```

Los puntos más importantes que hay que recordar con respecto a los parámetros verdaderos son:

- Los parámetros verdaderos que correspondan a los parámetros de entrada (de valor) del encabezado del procedimiento deben tener un valor antes de que se pueda invocar el procedimiento.
- Los parámetros verdaderos que correspondan a los parámetros de salida (variables) del encabezado del procedimiento deben haberse declarado como variables.

En el ejemplo anterior, *alt* y *base* deben tener algún valor antes de que se invoque el procedimiento, ya que actúan como sus datos de entrada. Además, es obligatorio dar una variable real como tercer parámetro verdadero de *triángulo*: no se permite una constante o expresión real como parámetro formal real que se declare como parámetro de salida (variable), ya que ninguna de las dos estará asociada a una localidad de memoria con nombre.

En seguida se escribirá un programa completo en Pascal que invocará dos veces al procedimiento *triángulo* autosuficiente, con diferentes parámetros verdaderos de entrada (de valor). En el programa principal se declararán las variables *alt*, *base* y *triárea* para el problema de área de triángulos. En realidad, se podrían utilizar los mismos nombres de variables que se usan en el procedimiento *triángulo*. Esto se debe a que las variables que se especifican en el encabezado del procedimiento son tan sólo parámetros formales que actúan como representantes de los datos de entrada y salida.

El programa *calcular* que se muestra a continuación calcula el área de dos triángulos diferentes y exhibe los resultados. Obsérvese que se incluye un procedimiento *impresión* para exhibir los resultados, lo que evita repetir la misma secuencia de proposiciones después de cada invocación del procedimiento. Tómese en cuenta además que el procedimiento *impresión* no tiene parámetros de salida

(variables) en la lista de parámetros formales, ya que este procedimiento no los necesita. En general, las declaraciones de procedimientos no se ven forzadas a contener parámetros (como en el caso de los procedimientos simples), o pueden contar solamente con parámetros de entrada (de valor), o tan sólo con parámetros de salida (variables), o finalmente con parámetros tanto de entrada (de valor) como de salida (variables). En lo que resta de este capítulo se estudiarán ejemplos de todos estos casos.

```

PROGRAMA calcular (input, output);
(* Calcular el área de dos triángulos diferentes *)
(* mediante un procedimiento con parámetros. *)
VAR
    alt,          (* altura de un triángulo *)
    base,         (* base de un triángulo *)
    triárea : real (* área de un triángulo *)
PROCEDURE triángulo (altura, base : real, VAR área : real);
(* Calcular el área de un triángulo a partir de su altura y su base *)
BEGIN
    área := 0.5 * altura * base
END;

PROCEDURE impresión (altura, base, área : real)
(* Exhibir altura, base y área de un triángulo *)
BEGIN
    writeln ('La altura del triángulo es ', altura:5:2);
    writeln ('La base del triángulo es ', base:5:2);
    writeln ('El área del triángulo es ', área:7:2)
END;

BEGIN (* programa principal *)
    (* Calcular el área del primer triángulo *)
    alt := 3;
    base := 4;
    triángulo (alt, base, triárea);
    impresión (alt, base, triárea);
    (* Calcular el área del segundo triángulo *)
    alt := 10;
    base := 6
    triángulo (alt, base, triárea);
    impresión (alt, base, triárea);
END.

```

## *Programa calcular*

La altura del triángulo es 3.00  
La base del triángulo es 4.00

El área del triángulo es 6.00  
La altura del triángulo es 10.00  
La base del triángulo es 6.00  
El área del triángulo es 30.00

---

*Salida del programa calcular*

## Variables locales y globales

Recuérdese que, en un procedimiento simple (sin parámetros), las variables o constantes a las que hace referencia el procedimiento se declaran en el programa principal. Estas variables se llaman *variables globales*, ya que cualquier procedimiento puede hacer referencia a ellas. Sin embargo, para que los procedimientos sean independientes del programa principal, es preciso transferir las variables en forma de parámetros. Dado que los procedimientos pueden ser programas autosuficientes, es posible declarar variables, constantes y hasta otros procedimientos dentro del procedimiento. Tales declaraciones son locales para el procedimiento. Las *variables locales* se declaran en un procedimiento y se conocen únicamente dentro de éste. Por ejemplo, considérese el siguiente procedimiento *datiempo* que tiene un parámetro de entrada (*tiempo*) y exhibe la hora de un reloj de 24 horas en términos de horas y minutos con un signo de dos puntos entre ellos. Así, si el parámetro verdadero de valor es el entero 1230, la salida con formato será

La hora es 12:30

---

```
PROCEDURE datiempo (tiempo : integer);
(* Exhibir la hora que se da en la forma "hhmm" en el formato      *)
(* de horas y minutos separados por un signo de dos puntos.          *)
CONST puntos = ':';           (* constante local                  *)
VAR horas, mins : integer;    (* variables locales                *)
BEGIN
  horas := tiempo DIV 100;
  mins := tiempo MOD 100;
  writeln ('La hora es ', horas:2, puntos, mins:2)
END;
```

---

### *Procedimiento datiempo*

En este procedimiento las variables *horas* y *mins* son variables locales que se conocen únicamente dentro del procedimiento. Además, *puntos* es una constante local. Más adelante, en el capítulo 7, se verá que el parámetro de entrada (de valor) *tiempo* también se puede considerar como variable local.

Este procedimiento es completamente autosuficiente e independiente de cualquier otro procedimiento y del programa principal, ya que se puede emplear cualquier valor para el parámetro verdadero. El único requisito es que concuerden los parámetros formales y verdaderos. De hecho, es posible declarar este procedimiento

(y cualquier otro que sea autosuficiente como éste) *exactamente* como se presenta aquí en cualquier programa principal, siempre que no se haya declarado previamente el nombre de procedimiento *datiempo* con algún otro propósito.

¿Qué sucederá si en el programa principal se declararon variables con los nombres *horas* y *mins*? En Pascal no importa si los nombres son idénticos. Aunque tengan los mismos identificadores, dentro del procedimiento se utilizarán, las variables locales, mientras que en el programa principal se usarán las variables globales. Dentro del procedimiento, las variables locales tienen *prioridad* sobre las variables globales y en el programa principal se desconocen. La ventaja obvia de lo anterior es que en un programa es posible declarar procedimientos escritos anteriormente con un mínimo de esfuerzo y (es de esperarse) sin modificaciones. En el capítulo 7 se estudiarán con mayor detalle las variables locales y globales.

## EJERCICIOS DE LA SECCIÓN 4.3

- 1 Identifiquense los parámetros de entrada (de valor) y de salida (variables) en el siguiente encabezado de procedimiento:

**PROCEDURE cheque (x, y, z : integer; VAR a, b, c : integer);**

- 2 ¿Cuál es la diferencia entre un parámetro formal y un parámetro verdadero?

- 3 Examíñese este encabezado de procedimiento:

**PROCEDURE inicio (tiempo, espacio:real; VAR día:real; signo:char);**

Supóngase que en el programa principal se hace la siguiente invocación del procedimiento:

**inicio (3.5, 6.0, hora, 'Z')**

Identifiquense los parámetros formales y los parámetros verdaderos. Después de ser invocado el procedimiento, ¿cuáles serán los valores de los parámetros formales que se especifican en el encabezado del procedimiento?

- 4 Examíñese el siguiente encabezado de procedimiento:

**PROCEDURE prueba (x, y : integer; VAR z : real);**

Encuéntrese el error en la siguiente invocación del procedimiento, si se supone que *tiempo* es una variable real:

**prueba (1, 2.0, tiempo)**

- 5 Examíñese el siguiente encabezado de procedimiento:

**PROCEDURE prueba (x, y : real; VAR z : real);**

Encuéntrese el error en la siguiente invocación del procedimiento:

**prueba (2.0, 3.0, 4.0)**

- 6 Examíñese el siguiente encabezado de procedimiento:

**PROCEDURE derecho (VAR z : real; x : char);**

Encuéntrese un error en la siguiente invocación del procedimiento; supóngase que *cero* es una variable real:

**derecho (cero,Z);**

Los ejercicios 7 a 12 se refieren al siguiente programa:

```
PROGRAM alcance (input, output);
VAR tum, num, temp : integer;
PROCEDURE prog (a, b : integer; VAR c : integer);
VAR reloj : integer
BEGIN
    reloj := a * b;
    reloj := reloj + 1;
    c := reloj + a;
    writeln (a, b, c, reloj)
END;
(* Programa principal *)
BEGIN
    tum := 1;
    num := 2;
    prog (tum, num, temp);
    writeln (temp);
    tum := 0;
    num := 1;
    prog (tum, num, temp);
    writeln (temp)
END.
```

- 7 Identifíquense las variables globales del programa principal.

- 8 Identifíquese la variable local declarada en el procedimiento *prog*.

- 9 Identifíquense los parámetros formales del procedimiento *pro*. Determinense los parámetros de entrada (de valor) y de salida (variables).

- 10 Determiníense los parámetros de valor verdaderos en la primera invocación del procedimiento.
- 11 Determiníense los parámetros de valor verdaderos en la segunda invocación del procedimiento.
- 12 Determíñese la salida del programa. ¿Cuántos renglones se exhibirán?

## SECCIÓN 4.4 RESOLUCIÓN DE PROBLEMAS MEDIANTE PROCEDIMIENTOS CON PARÁMETROS

En esta sección se aplicarán procedimientos con parámetros a la resolución de problemas. En particular, se incluirá un análisis detallado de los tres componentes principales de la resolución de problemas: análisis de procedimientos, resolución del problema y resolución en computadora. Considérese el siguiente problema:

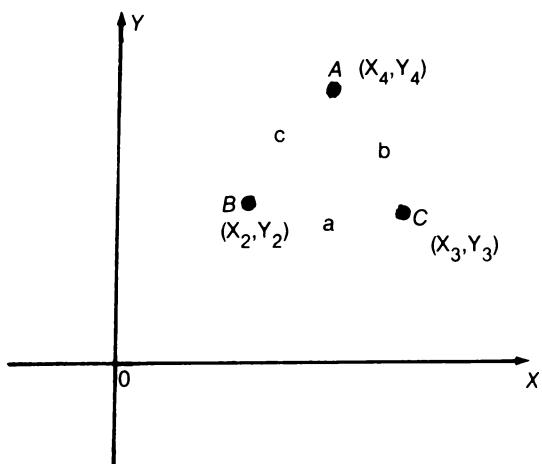
### Problema 4.2

*Los datos de entrada incluyen las coordenadas en el plano (coordenadas cartesianas) de los tres vértices de un triángulo como números reales. Calcúlese y exhibase el área del triángulo representado por estos datos.*

### Análisis del problema

La figura 4-7 muestra un conjunto de tres puntos que pueden ser los datos de entrada. Cada uno posee un valor  $x$  y un valor  $y$ . Se dibujaron las aristas apropiadas entre los puntos para formar un triángulo. Los tres lados se etiquetaron con las letras  $a$ ,  $b$  y  $c$ . Los vértices del triángulo se representan mediante las letras mayúsculas  $A$ ,  $B$  y  $C$ . Se proporcionan datos de entrada que incluyen las coorde-

**Figura 4-7** Triángulo especificado por vértices en un plano.



nadas  $x$  y  $y$  de cada vértice:  $A = (x_1, y_1)$ ,  $B = (x_2, y_2)$  y  $C = (x_3, y_3)$ . Dadas las coordenadas de los vértices, es posible calcular la longitud de los tres lados mediante la fórmula para la distancia entre dos puntos (del teorema de Pitágoras, en geometría analítica). Por ejemplo, la distancia entre los puntos representados por  $(x_1, y_1)$  y  $(x_2, y_2)$  está dada por la fórmula

$$\text{Distancia} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Ahora, ya que es posible calcular la longitud de los tres lados, es preciso calcular el área del triángulo. Por fortuna, existe una fórmula en geometría (la fórmula de Heron) que dice: dadas las longitudes de los tres lados de un triángulo,  $a$ ,  $b$  y  $c$ , el área se calcula mediante

$$\text{Área} = \sqrt{s(s - a)(s - b)(s - c)},$$

donde  $s$  es la mitad del perímetro del triángulo. Es decir,

$$s = (a + b + c) / 2.$$

Se resolverá a mano un ejemplo específico. Supóngase que las coordenadas  $x$  y  $y$  de los vértices son las siguientes:

$$A = (0,0) \quad B = (0,3) \quad C (4,0)$$

Las longitudes de los tres lados del triángulo formados por los vértices  $A$ ,  $B$  y  $C$  son

$$a = 3 \quad b = 4 \quad c = 5.$$

El valor  $s$  (la mitad del perímetro) en este caso es

$$s = (3 + 4 + 5)/2 = 6.$$

Por tanto, el área es

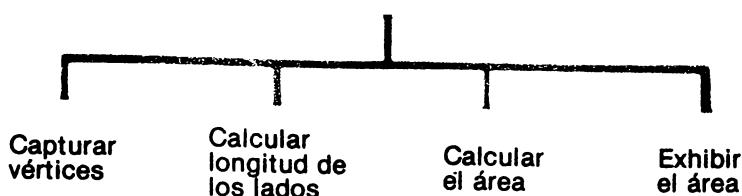
$$\begin{aligned}\text{Área} &= \sqrt{6(6 - 3)(6 - 4)(6 - 5)} \\ &= \sqrt{36} \\ &= 6.\end{aligned}$$

Ya se analizó el problema lo suficiente como para poder hacer el diseño descendente (véase la Fig. 4-8).

## Diseño descendente

Este problema se puede dividir en cuatro subproblemas:

**Problema  
de área  
de triángulos**



**Figura 4-8** Diseño descendente del problema de área de triángulos.

SUBPROBLEMA 1: Obtener las coordenadas de los vértices.

SUBPROBLEMA 2: Calcular las longitudes de los tres lados.

SUBPROBLEMA 3: Calcular el área del triángulo.

SUBPROBLEMA 4: Exhibir el área del triángulo.

Puesto que es menester obtener tres conjuntos de coordenadas, se escribirá un procedimiento llamado *vértice* para capturar una pareja de coordenadas. Además, es preciso calcular la longitud de los tres lados, por lo que sería conveniente tener un procedimiento llamado *longitud* para calcular la distancia entre dos puntos dados. Por último se escribirá un procedimiento llamado *calcárea* para calcular el área de un triángulo mediante la fórmula de Heron en caso de que sea necesario realizar este cálculo más de una vez.

### Algoritmo

A continuación es preciso refinar los subproblemas y escribir el algoritmo. El seudocódigo del algoritmo quedaría así:

#### Seudocódigo del problema 4.2

PASO 1: Obtener las coordenadas de los tres vértices del dispositivo de entrada mediante la llamada del procedimiento de entrada (*vértice*) tres veces con diferentes parámetros verdaderos.

PASO 2: Calcular la longitud de cada lado del triángulo llamando tres veces al procedimiento apropiado (*longitud*) con los parámetros adecuados.

PASO 3: Calcular el área del triángulo mediante la fórmula de Heron.

PASO 4: Exhibir el área del triángulo.

### Resolución en computadora

Ahora se puede comenzar a escribir los procedimientos para resolver cada uno de los subproblemas. El procedimiento para obtener un vértice de los datos de entrada, llamado *vértice* tendrá dos parámetros de salida (variables). Se llamará a estos parámetros *x* y *y* para representar las coordenadas *x* y *y* del punto que se va a pro-

```
PROCEDURE vértice (VAR x, y : real);
(* Procedimiento para capturar las coordenadas x y de un punto *)
BEGIN
    write ('Por favor escriba la coordenada x del punto:');
    readln (x);
    write ('Por favor escriba la coordenada y del punto:');
    readln (y);
    writeln('Las coordenadas son:', x, y)
END;
```

---

### *Procedimiento vértice*

Ahora hay que escribir un procedimiento para calcular la distancia entre dos puntos dados,  $(x_1, y_1)$   $(x_2, y_2)$  por ejemplo. En este caso las coordenadas son parámetros de entrada (de valor). La distancia, representada por el parámetro formal *distancia*, es un parámetro de salida (variable), ya que es preciso transferir este resultado al programa principal. Tómese nota del uso adicional de variables locales y la función incluida de raíz cuadrada (*sqr*) en el procedimiento. No es preciso declarar la función *sqr*, ya que está predeclarada en Pascal.

---

```
PROCEDURE longitud (x1, y1, x2, y2 : real; VAR distancia : real);
(* Determinar la distancia entre los puntos (x1, y1) y *)
(* (x2, y2) y colocar el resultado en "distancia". *)
.VAR
    cambiox,          (* cuadrado del cambio en x *)
    cambioy : real;   (* cuadrado del cambio en y *)
BEGIN
    (* calcular los términos de la expresión de raíz cuadrada *)
    cambiox := sqrt (x1 — x2);
    cambioy := sqr (y1 — y2);
    (* calcular "distancia" mediante la fórmula de la distancia *)
    distancia := sqrt (cambiox + cambioy)
END;
```

---

### *Procedimiento longitud*

Por último, es menester escribir un procedimiento llamado *calcárea* para determinar el área del triángulo mediante la fórmula de Heron. Una vez más, se emplean variables locales para almacenar el perímetro del triángulo. Los parámetros de entrada (de valor) son las longitudes de los tres lados *a*, *b* y *c* y el parámetro de salida (variable) es el área del triángulo.

---

```

PROCEDURE calcárea (a, b, c : real; Var área : real);
(* Determinar el área de un triángulo de lados a, b y c *)
(* y colocar el resultado en "área". *)
VAR
    s,                      (* mitad del perímetro del triángulo *)
    perímetro : real;       (* perímetro del triángulo *)
BEGIN
    (* Calcular el perímetro y la mitad del perímetro del triángulo *)
    perímetro := a + b + c;
    s := perímetro /2;
    (* Calcular el área del triángulo mediante la fórmula de Heron *)
    área := sqrt (s * (s-a) * (s-b) * (s-c))
END;

```

---

*Procedimiento calcárea*

La figura 4-7 muestra la relación entre las variables que se usan en el programa y su interpretación física. En seguida se completará el programa al combinar los procedimientos con el programa principal. A continuación se muestra el programa Heron seguido de la exhibición que resultaría de una posible ejecución.

---

```

PROGRAM Heron (input, output);
(* Determinar el área de un triángulo dadas las      *)
(* coordenadas de los vértices. Se empleará la      *)
(* fórmula de Heron para el cálculo del área      *)
VAR
    x1, y1,                      (* coordenadas del primer vértice *)
    x2, y2,                      (* coordenadas del segundo vértice *)
    x3, y3,                      (* coordenadas del tercer vértice *)
    ladoa,                         (* longitud del lado a *)
    ladob,                         (* longitud del lado b *)
    ladoc,                         (* longitud del lado c *)
    área : real;                  (* área del triángulo *)

```

---

```

PROCEDURE vértice (VAR x, y : real);
(* Procedimiento para capturar las coordenadas x, y y de un punto *)
BEGIN
    write ('Por favor escriba la coordenada x del punto: ');
    readln (x);
    write ('Por favor escriba la coordenada y del punto: ');
    readln (y);
    writeln ('Las coordenadas son: ', x, y)
END;

```

---

```

PROCEDURE longitud (x1, y1, x2, y2 : real; VAR distancia : real);
(* Determinar la distancia entre los puntos (x1, y1) y *)
(* (x2, y2) y colocar el resultado en "distancia".

```

BEGIN

cambiox, (\* cuadrado del cambio en x\*)  
 cambioy : real; (\* cuadrado del cambio en y\*)

(\* Calcular los términos de la expresión de raíz cuadrada \*)  
 cambiox : = sqr (x1 — x2);  
 cambioy : = sqrt (y1 — y2);  
 (\* Calcular "distancia" mediante la fórmula de la distancia \*)  
 distancia : = sqrt (cambiox + cambioy)

END;

PROCEDURE calcárea (a, b, c, : real; VAR área : real);

(\* Determinar el área de un triángulo de lados a, b y c \*)

(\* y colocar el resultado en "área". \*)

VAR

s, (\* mitad del perímetro del triángulo\*)  
 perímetro : real; (\* perímetro del triángulo \*)

BEGIN

(\* Calcular el perímetro y la mitad del perímetro del triángulo \*)  
 perímetro := a + b + c;

s := perímetro / 2;

(\* Calcular el área del triángulo mediante la fórmula de Heron \*)

área := sqrt (s \* (s-a) \* (s-b) \* (s-c))

END;

(\* Programa principal del problema 4.2 \*)

BEGIN

(\* Capturar las coordenadas de los tres vértices \*)

vértice (x1, y1);

vértice (x2, y2);

vértice (x3, y3);

(\* Determinar la longitud de los lados y guardar \*)

(\* los resultados en ladoa, ladob y ladoc. \*)

longitud (x1, y1, x2, y2, ladoa);

longitud (x1, y1, x3, y3, ladob);

longitud (x2, y2, x3, y3, ladoc);

(\*Determinar el área del triángulo \*)

calcárea (ladoa, ladob, ladoc, área);

(\* Exhibir el área calculada \*)

writeln ('El área del triángulo es ', área:7:2)

END.

*Programa Heron*

Por favor escriba la coordenada x del punto: 0

Por favor escriba la coordenada y del punto: 0

Por favor escriba la coordenada x del punto: 0

Por favor escriba la coordenada y del punto: 3

Por favor escriba la coordenada x del punto: **4**  
 Por favor escriba la coordenada y del punto: **0**  
 El área del triángulo es **6.00**

-----  
*Programa Heron: ejemplo de ejecución*

## EJERCICIOS DE LA SECCIÓN 4.4

- 1 Determíñese la salida exacta del siguiente programa.
- ```

PROGRAM principal (input, output);
VAR a, b, c : integer;
PROCEDURE sub (x : integer);
VAR a, b : integer;
BEGIN
  a := 1;
  b := 2;
  x := a + b;
  writeln (a, b, x)
END;
BEGIN (* Programa principal *)
  a := 3;
  b := 5;
  c := 4;
  sub (c);
  writeln (a, b, c)
END.

```
- 2 Escríbase un procedimiento llamado *cubo* que tenga un parámetro de entrada (de valor) llamado *longitud* y un parámetro de salida (variable) llamado *volumen*. El procedimiento deberá calcular el volumen del cubo cuya arista sea de la longitud especificada. Todas las variables son reales.
  - 3 Escríbase un procedimiento llamado *magnitud* que tenga dos parámetros de entrada (de valor) llamados *x1* y *x2* y un parámetro de salida (variable) llamado *distancia*. El procedimiento deberá calcular la distancia absoluta entre los puntos *x1* y *x2*. Todas las variables son reales.
  - 4 Escríbase un procedimiento llamado *dígito* que tenga un parámetro de entrada (de valor) llamado *número* y un parámetro de salida (variable) llamado *másbajo*. El procedimiento deberá extraer el dígito de unidades del valor *número* y colocarlo en *másbajo*. Por ejemplo, si *número* es 234, el valor que se almacenará en *másbajo* será cuatro, el dígito de unidades. Si *número* es 1200, el valor almacenado en *másbajo* será cero. (Sugerencia: utilícese el operador MOD.)
- \* Considérese el siguiente problema. Un piso rectangular que mide 12 metros por 15 metros está cubierto parcialmente por tapetes circulares. El radio de un

tapete es un metro. El radio del otro es dos metros. Los ejercicios 5, 6 y 7 se ocupan de hallar el área de la parte del piso que está descubierta.

- 5 Escríbase un procedimiento con dos parámetros formales que determine el área de un círculo. El primer parámetro formal es el radio y el segundo es el área correspondiente. Supóngase que ambos son reales.
- 6 Escríbase un procedimiento con tres parámetros formales que calcule el área de un rectángulo de longitud y anchura dadas.
- 7 Escríbase un programa completo en Pascal que incluya los procedimientos desarrollados en los ejercicios 5 y 6. El programa deberá incluir invocaciones de los procedimientos para calcular el área de cada círculo y una invocación de procedimiento para determinar el área del piso rectangular. Después, el programa principal deberá determinar el área descubierta del piso y exhibir ese resultado.

## SECCIÓN 4.5 DISEÑO Y PRUEBA DESCENDENTES

Es frecuente que la parte mas difícil del diseño descendente sea tomar las decisiones referentes a la división del trabajo. Al comenzar un diseño no siempre es posible ver los problemas que pueden surgir durante la resolución de un problema de un nivel inferior. Los problemas se deben casi siempre a dos razones: la división del trabajo requerido para la resolución del programa no es la apropiada o las estructuras de datos elegidas (la forma como se van a almacenar y procesar los datos) no son adecuadas. Este último problema se analizará con mayor detalle en un capítulo posterior. El primer problema, la división poco apropiada del trabajo, se puede atribuir muchas veces a que no se comprende perfectamente el problema que se va a resolver.

Sin importar cuál sea la causa del problema, los errores de diseño no se deben considerar como dificultades menores que se deben corregir y olvidar o “defectos” que se deben ocultar. Si existen dificultades, es preciso analizar cuidadosamente cada problema de diseño para identificar su causa. Si se utiliza esta técnica, el diseñador estará menos propenso a cometer los mismos errores cuando se presenten problemas similares en otras tareas de diseño.

A continuación, mediante un ejemplo, se indicarán los primeros niveles de subdivisión en un problema de cierta magnitud. Al hacer esto, se espera que el lector se convencerá de que los problemas grandes realmente se pueden descomponer en partes más sencillas. La solución incluye algunos ejemplos de cómo se toman decisiones en los programas en Pascal. Las primeras construcciones para la toma de decisiones se estudiarán con detalle en el capítulo 5.

### Revisión de ortografía

Las organizaciones que procesan grandes cantidades de textos emplean a menudo programas de revisión de ortografía. Este tipo de programas trata de identificar todas las palabras mal escritas de un fragmento de texto y exhibe el número del

renglón en el que aparece la palabra mal escrita. El programa suele contar con una lista de palabras correctamente escritas (un diccionario) a la que hace referencia.

La primera versión de la solución es como sigue:

PASO 1: Leer el texto.

PASO 2: Encontrar todas las palabras mal escritas e imprimirlas junto con sus números de renglón.

Conviene analizar un poco el problema antes de avanzar demasiado pronto al siguiente paso. Es necesario determinar el tamaño máximo del texto que constituye la entrada del programa. Si no se puede determinar un tamaño máximo, o si el tamaño máximo es mayor que la memoria disponible para almacenamiento de datos en el sistema de cómputo, no será posible completar el paso 1. Para evitar estos problemas, podría ser mejor leer únicamente una porción del texto, procesarlo y repetir estos pasos hasta que se haya procesado todo el texto. La versión modificada podría ser entonces:

PASO 1 Leer un fragmento de texto de tamaño apropiado.

PASO 2 Encontrar todas las palabras mal escritas en el fragmento que se acaba de leer y exhibirlas junto con sus números de renglón.

PASO 3 Si queda texto por procesar, volver al paso 1.

Ya es posible ahora refinar cualquiera de los tres pasos de la primera versión modificada. Para refinar el paso 1 se debe decidir qué es un “fragmento de tamaño apropiado”. Existen muchas opciones razonables, pero en este análisis se supondrá que el fragmento que ha de leerse consiste en un solo renglón. Esto ayudará también a mantener un registro de los números de renglón en que aparecen las palabras mal escritas. He aquí, pues, la refinación del paso 1 (recuérdese que se va a procesar un renglón a la vez):

PASO 1.1 Ajustar el número de palabras a cero.

PASO 1.2 Si se llegó al final del renglón, pasar al siguiente (recuérdese la forma como tratan al fin de línea las proposiciones *readln*) y proceder con el paso 1.4.

PASO 1.3 Leer la siguiente palabra, incrementar el número de palabras leídas y volver al paso 1.2.

PASO 1.4 Incrementar el número de renglón. (Se comenzará con el número de renglón igual a cero.)

El paso 2 es la médula del revisor de ortografía. Se supone que es posible buscar una palabra determinada en el diccionario, lo que dará lugar a una indicación de que se encontró la palabra o bien de que no se encontró. Esta parte del programa no se mostrará aquí. Una ventaja del procedimiento de desarrollo descendente es que permite suponer que una tarea determinada se puede hacer y que es posible desarrollar el resto del programa, aunque el procedimiento que lleve a cabo esa tarea no se haya terminado aún. He aquí la refinación del paso 2.

**PASO 2.1** Repetir los pasos 2.2 a 2.4 para cada una de las palabras obtenidas en el paso 1.

**PASO 2.2** Buscar la palabra actual en el diccionario.

**PASO 2.3** Si se encuentra la palabra (estaba bien escrita), no es preciso realizar acción alguna.

**PASO 2.4** Si no se encuentra la palabra (estaba mal escrita), exhibir el número de renglón y la palabra.

El paso 3 es muy sencillo y no se refinará más en este análisis.

Ahora se ejecutará el programa en su forma actual. (Se recomienda al lector hacer esto siempre con sus programas para convencerse de que trabaja como se esperaba. ¡No es muy probable que un programa en Pascal falle si se revisó exhaustivamente durante el desarrollo!)

Supóngase que los datos de entrada constan de varios renglones de texto como sigue:

Toda fórmula susceptible de cálculo en la máquina analítica contiene ciertas operaciones algebraicas que deben realizarse sobre determinadas letras, y algunas otras modificaciones que dependen del valor numérico asignado a dichas letras.<sup>1</sup>

Aunque no se ha expresado de manera explícita, es evidente ahora que el programa debe tomar en cuenta la puntuación, las palabras separadas por guiones y varias formas, como los plurales. No es preciso ocuparse de ello en este momento, pero las refinaciones posteriores deberán resolver esos problemas.

El paso 1 procesará el primer renglón de texto:

Toda fórmula susceptible de cálculo en la máquina analítica

y producirá la siguiente lista de palabras (que se han escrito en minúsculas por uniformidad):

toda  
fórmula  
susceptible  
de  
cálculo  
en  
la  
máquina  
analítica

Después de leer la palabra analítica del renglón de datos de entrada se descubre que ya no hay más palabras, por lo que se avanza al paso 2 para procesar estas nueve palabras, siendo el número de renglón igual a uno.

<sup>1</sup>Charles Babbage and His Calculating Engines, Philip y Emily Morrison, editores. Dover Publications, Nueva York, 1961.

El paso 2 intentará en primer lugar localizar la palabra *toda* en el diccionario. Puesto que está bien escrita y es probable que se encuentre en la mayor parte de los diccionarios, no se producirá salida alguna. La palabra *fórmula* se verifica con resultados similares. Se revisan las demás palabras del renglón, una tras otra, hasta que el paso 2 llega a la palabra *analítica*. Aunque esta palabra está bien escrita, supóngase que no está en el diccionario. Puesto que se partió del supuesto de que todas las palabras correctamente escritas están en el diccionario, se dará por sentado que *analítica* está mal escrita. (La mayor parte de los revisores de ortografía más completos permiten el uso de diccionarios suplementarios y la modificación de los diccionarios existentes.) Por tanto, el paso 2.4 exhibirá un mensaje que podría ser “Renglón 1, analítica.”

Una vez procesadas todas las palabras del primer renglón, el paso 3 determinará si existen más renglones. Puesto que sí existen, se volverá al paso 1 para procesar el siguiente renglón. En un momento dado, el paso 3 determinará que se agotaron los datos de entrada y el programa terminará.

## Prueba descendente

Es preciso darse cuenta desde el principio que el hecho de probar un programa no garantiza que nunca va a fallar. La prueba sólo puede servir para demostrar la presencia de errores, no su ausencia.

De cualquier manera, sería imprudente suponer que un programa funciona correctamente sin realizar algún tipo de pruebas, ya que una prueba cuidadosa suele detectar la mayor parte de los errores. Aquí se examinarán dos aspectos importantes de la prueba de programas: la selección de datos de prueba y la prueba descendente y ascendente de un programa.

Al realizar la selección de datos de prueba se debe tener en mente la aplicación del programa. En muchos casos, lo mejor es obtener los datos de prueba de las personas que van a usar el programa cuando esté completo. Esto tiene dos ventajas. En primer lugar, los datos de prueba estarán apegados a la realidad, ya que son una muestra de los datos reales. En segundo lugar, se localizarán posibles fallas en la documentación de lo que espera el programa como datos de entrada.

Por ejemplo, supóngase que se tiene un programa que requiere un número de tres dígitos en las columnas 1 a 3 del renglón de datos de entrada. Un usuario podría creer que las primeras tres columnas pueden tener cualquier número, con espacios en blanco a la izquierda, a la derecha o en medio, y preparar los datos de prueba con esa suposición. Puesto que es probable que el programa falle si no se preparan adecuadamente los datos, se habrá localizado a tiempo un problema y se podrá corregir antes de que el programa pase a la fase de producción.

Por cierto, la mayor parte de los programas buenos que se producen verifican en forma exhaustiva los datos de entrada para garantizar que cumple con los requisitos. No es raro encontrar que más de la mitad del código de un programa se ocupa de la validación de los datos.

Además de probar datos representativos, es importante probar casos excepcionales para garantizar que los datos poco comunes no provoquen una falla del programa.

La prueba incremental de un programa puede avanzar en dos direcciones, a veces simultáneamente. La prueba *descendente* se lleva a cabo mediante la ejecución de los procedimientos en la parte superior del árbol de desarrollo, mientras que la prueba *ascendente* conlleva la ejecución de los procedimientos de bajo nivel antes de juntarlos con otros.

Puesto que la prueba descendente se realiza muchas veces antes de que se escriban todos los procedimientos, es necesario engañar al programa para que crea que está completo. De hecho, al analizar el programa de revisión de ortografía, se supuso la existencia de un procedimiento correcto para obtener la siguiente palabra de los datos de entrada y de un procedimiento para determinar si una palabra se encuentra en un diccionario. Para realizar este truco se utilizan procedimientos muy simples llamados *instrumentales*. Un procedimiento instrumental se comporta exactamente como el “verdadero”, con la excepción de que produce resultados que no son necesariamente correctos, pero sí están predeterminados. Los procedimientos instrumentales también permiten verificar si concuerdan los parámetros formales y los verdaderos.

Se analizará el uso de procedimientos instrumentales en la prueba descendente del revisor de ortografía. En vez de producir un procedimiento que obtenga la siguiente palabra de un renglón de datos de entrada (cuidando la puntuación, separación por guiones, etc.) se podría producir un procedimiento instrumental que simplemente proporcione una palabra de una lista previamente preparada y que incluya palabras bien escritas y mal escritas. (Adviértase que es preciso saber qué es lo que hace el procedimiento instrumental para poder interpretar la salida.)

También se podría escribir un procedimiento instrumental para la búsqueda en el diccionario. En vez de procesar un diccionario completo (que normalmente requiere técnicas de búsqueda avanzadas), se podría simplemente buscar en una lista breve de palabras bien escritas aquellas que podrían producir el procedimiento instrumental *que obtiene una palabra*.

Mediante estos dos procedimientos instrumentales y el resto del programa es posible hacer que el conjunto funcione correctamente. A continuación se podría escribir el procedimiento *verdadero* para capturar las palabras de los datos de entrada y conservar el procedimiento instrumental para la búsqueda en el diccionario. Si hay fallas, se sabrá que el problema radica en el procedimiento de captura y no en el resto del código que ya se probó. Por último, se sustituirá el procedimiento instrumental de búsqueda en el diccionario por el procedimiento verdadero.

La prueba ascendente se realiza en la dirección opuesta. Es decir, se prueban primero las soluciones individuales de los subproblemas más pequeños y después se combinan para producir la solución del problema completo. Esta forma de prueba requiere la escritura de programas principales breves para probar cada una de las soluciones a los subproblemas. Además, será necesario probar estas soluciones mediante un programa principal para garantizar que funcionen bien juntas.

Si bien algunos programadores podrían optar exclusivamente por la prueba ascendente o la descendente, en la mayor parte de los casos se emplea una combinación de ambas técnicas. Sea cual sea la que elija el lector, deberá seleccionar cuidadosamente los datos de prueba, ya que es la única forma de obtener una solución que se acerque a la perfección.

## SECCIÓN 4.6 TÉCNICAS DE PRUEBA Y DEPURACIÓN

Si se escribió un procedimiento con parámetros que constituye un programa autosuficiente, será preciso probarlo con distintos datos de entrada. La estrategia se parece a la prueba de programas que se analizó en el capítulo anterior. Empero, como es menester transferir los parámetros verdaderos, se puede simular o imitar la ejecución del programa principal mediante el suministro de valores verdaderos al procedimiento en forma de casos de prueba.

Una estrategia de prueba común consiste en insertar proposiciones *read* y *write* como ayuda para la depuración. Estas proposiciones *read* y *write* se pueden eliminar sin peligro una vez depurado el procedimiento.

En seguida se estudiará un ejemplo específico. He aquí un procedimiento que calcula el promedio de tres enteros:

```
PROCEDURE prom (núm1, núm2, núm3 : integer;
                VAR promedio : real);
VAR total : inter;
BEGIN
    total := núm1 + núm2 + núm3;
    promedio := total / 3
END;
```

Para probar este procedimiento se puede escribir un programa que lea los tres números y transfiera esos valores al procedimiento. Se puede insertar una proposición *write*, ya sea en el procedimiento o en el programa principal, para verificar si funciona bien el procedimiento. El siguiente es un ejemplo de programa de prueba que incluye una proposición *write* en el procedimiento para su depuración. Esta proposición se eliminará después de depurar el procedimiento.

```
PROGRAM prueba (input, output);
(* Programa para probar un procedimiento *)
VAR
    valor1, valor2, valor3 : integer;
    medio : real;
PROCEDURE prom (núm1, núm2, núm3 : integer; VAR promedio : real);
VAR total : integer;
BEGIN
    total := núm1 + núm2 + núm3;
    promedio := total / 3
    (* PROPOSICIÓN WRITE PARA DEPURACIÓN *)
    writeln ('El promedio del caso de prueba es ', promedio:7:2)
END;
(* Programa principal *)
BEGIN
    write ('Escriba tres números de prueba: ');
    readln (valor1, valor2, valor3);
```

### *Programa prueba*

Obsérvese que la proposición *write* para depuración se podría haber insertado en el programa principal después de la invocación del procedimiento. En este caso el parámetro verdadero (de salida) en el programa principal se llama *medio*. Así, la proposición *write* para depuración podría haber sido la siguiente:

writeln ('El promedio del caso de prueba es ', medio:7:2)

Cuando los problemas son más complejos y los programas se vuelven difíciles de manejar, el uso de proposiciones *write* para depuración será aún más valioso en la fase de prueba y depuración.

La siguiente lista incluye varios puntos que conviene recordar cuando se prueban y depuran procedimientos

## RECORDATORIOS DE PASCAL

- Los procedimientos se deben declarar y se colocan después de las declaraciones de constantes y variables pero antes de las proposiciones ejecutables del programa principal.
- Para invocar un procedimiento se escribe su nombre seguido de la lista de parámetros verdaderos entre paréntesis (en caso de requerirse).
- Las proposiciones ejecutables de un procedimiento se colocan entre las palabras reservadas BEGIN y END.
- Todo procedimiento debe terminar con un signo de punto y coma.
- El tipo y orden de los parámetros verdaderos deben concordar con los que se especifiquen para los parámetros formales correspondientes.
- Los parámetros verdaderos que correspondan a los parámetros de entrada (de valor) deben tener un valor antes de invocarse el procedimiento.
- Los parámetros verdaderos que corresponden a los parámetros de salida (variables) se deben declarar como variables.
- Los parámetros de entrada (de valor) no pueden cambiar los valores de los parámetros verdaderos en el programa principal, pero los parámetros de salida (variables) sí permiten cambiar los valores de los parámetros verdaderos correspondientes.
- Las variables declaradas en el programa principal son globales y cualquier procedimiento en el que no se vuelvan a declarar puede hacer referencia a ellas.
- Las variables que se declaran en un procedimiento son locales. Se conocen únicamente dentro de ese procedimiento y tienen prioridad sobre las variables con los mismos identificadores en el programa principal.

**SECCIÓN 4.7 REPASO DEL CAPÍTULO**

Mediante el diseño descendente es posible dividir un problema complejo en subproblemas. Éstos a su vez se refinan sucesivamente hasta hacer posible la resolución en computadora. En la computadora, las soluciones de los subproblemas se pueden expresar en términos de subprogramas llamados procedimientos. Los procedimientos en Pascal tienen una estructura que imita a la de los programas en Pascal:

```
PROCEDURE nombre (lista de parámetros);
CONST declaraciones;
VAR declaraciones;
BEGIN
    proposición;
    proposición;
    ...
    proposición
END;
```

Los procedimientos que se escriben en forma de módulos de programa independientes y autosuficientes pueden resolver en forma efectiva problemas complejos en la computadora. Así, es posible probar y depurar los procedimientos de manera independiente. Los cambios a los procedimientos se pueden realizar más eficientemente cuando los procedimientos se escriben en forma de subprogramas autosuficientes.

La información se transfiere entre los procedimientos y el programa principal mediante objetos llamados parámetros. Los parámetros formales se especifican en la lista de parámetros del encabezado del procedimiento. Los parámetros formales pueden ser de entrada (de valor) o de salida (variables). Los parámetros verdaderos se especifican en la proposición de invocación del procedimiento.

En seguida se proporciona un resumen detallado de los procedimientos en Pascal, el cual se puede usar como referencia en el futuro.

**REFERENCIAS DE PASCAL****1 Procedimientos**

- 1.1 *Encabezado del procedimiento*: incluye el nombre del procedimiento y una lista de parámetros opcional entre paréntesis.
- 1.2 Los procedimientos se colocan después de las declaraciones de constantes y variables del programa principal y antes de las proposiciones ejecutables del programa principal.
- 1.3 Para invocar un procedimiento se escribe su nombre seguido de la lista entre paréntesis de parámetros verdaderos que concuerda en orden y alcance con la lista de parámetros formales.

**2 Parámetros**

- 2.1 *Parámetros formales*: se especifican en la lista de parámetros del encabezado del procedimiento.

- 2.2 *Parámetros de entrada (de valor)*: proporcionan valores de entrada al procedimiento.
- 2.3 *Parámetros de salida (variables)*: deben comenzar con la palabra reservada VAR en la lista de parámetros y son la forma de comunicar resultados del procedimiento a las variables correspondientes en la invocación del procedimiento.

Ejemplo:

```
PROCEDURE nombre (a, b, c : real; VAR x, y : real);
```

Así, *a*, *b* y *c* son parámetros de entrada (de valor) y *x* y *y* son parámetros de salida (variables).

### 3 Variables

- 3.1 *Variables globales*: se declaran en el programa principal y se hace referencia a ellas en los procedimientos en los que no se vuelven a declarar.
- 3.2 *Variables locales*: se declaran en un procedimiento y se conocen únicamente dentro de ese procedimiento.

### 4 Ejemplo de estructura de un procedimiento dentro de un programa en Pascal:

```
PROGRAM ejemplo (inpunt, output);
CONST declaraciones; (* globales *)
VAR declaraciones; (* globales *)
PROCEDURE nombre (lista de parámetros);
CONST declaraciones; (* locales *)
VAR declaraciones; (* locales *)
BEGIN
    proposición;
    proposición;
    ...
    proposición
    END;
(* Programa principal *)
BEGIN
    proposición;
    nombre (lista de parámetros verdaderos);
    (* invocación del procedimiento *)
    ...
    proposición
    END.
```

## Palabras clave del capítulo 4

impulsor	procedimiento
invocación	procedimiento instrumental
parámetro	programa principal
parámetro de entrada (de valor)	prueba ascendente
parámetro de salida (variable)	prueba descendente
parámetro formal	variable global
parámetro verdadero	variable local

¿Puede tomar decisiones la computadora? En el siguiente capítulo se estudiará una construcción importante de los programas que sirve para tomar decisiones y elegir diferentes cursos de acción. En particular, se analizarán las proposiciones IF-THEN-ELSE y CASE en Pascal. Estas proposiciones permiten resolver problemas más complejos en los que se debe optar por diferentes acciones según si se cumple o no alguna condición.

## EJERCICIOS DEL CAPÍTULO 4

### ★ EJERCICIOS ESENCIALES

- 1 Ya se analizaron dos tipos de parámetros que se pueden emplear con los procedimientos en Pascal: parámetros de entrada (de valor) y de salida (variables). ¿Existe también algún parámetro de actualización que pueda emplearse para proporcionar datos de entrada a un procedimiento y mediante el cual pueda obtenerse algún valor (resultado)?
- 2 Dados dos tiempos arbitrarios en la escala de 24 horas, representados mediante enteros de la forma *hhmm*, escribase un procedimiento *tsuma* que produzca (como parámetro de salida) la suma de estos tiempos en forma de un entero del tipo *hhmm* donde *d* representa al número de días.
- 3 Supóngase que las variables enteras *a* y *b* representan un número racional *a/b*. Escribase un procedimiento llamado *sumrac* que obtenga la suma de dos números racionales de este tipo. Es decir, encuéntrense los enteros *e* y *f* tales que

$$\frac{e}{f} = \frac{a}{b} + \frac{c}{d}$$

donde *a*, *b*, *c*, *d*, *e* y *f* son enteros. No es necesario reducir la fracción *e/f* a su forma más simple.

- 4 Escribase un procedimiento llamado *conv3* que convierta tres caracteres, cada uno de los cuales contiene un dígito decimal, al entero equivalente. Por ejemplo, si los parámetros de entrada son '1', '2' y '3', entonces el entero resultante será 123.
- 5 El Pascal no incluye funciones estándar para la cotangente, secante o cosecante. ¿Por qué no causa problemas esta omisión?

### ★ ★ EJERCICIOS IMPORTANTES

- 6 El Pascal estándar permite utilizar procedimientos externos previamente compilados. Uno de estos procedimientos que se encuentra en muchos sistemas proporciona la hora actual. Si el sistema del lector está en este caso, escribase un programa en Pascal que, al ejecutarse, exhiba la hora actual.

- 7 ¿Podría utilizarse un procedimiento de hora actual como el mencionado en el ejercicio o para determinar el tiempo de ejecución total de un programa? ¿Qué sucedería si se ejecutara varias veces un programa de medición del tiempo como ése en un sistema de cómputo de varios usuarios?
- 8 ¿Puede el lector hallar una forma de determinar el valor de una variable local tan pronto como se ejecute por segunda vez el procedimiento en el cual se declara?
- 9 La solución de un sistema de ecuaciones lineales simultáneas con dos incógnitas se puede obtener fácilmente mediante la regla de Cramer. Supóngase que el sistema de ecuaciones es

$$ax + by = c \quad y \quad dx + ey = f.$$

La regla de Cramer dice que, si existe una solución,

$$x = \frac{ce - fb}{ae - db} \quad y \quad y = \frac{af - dc}{ae - db}.$$

Escríbase un procedimiento en Pascal con parámetros de entrada  $a, b, c, d, e$  y  $f$  que determine la solución de las ecuaciones simultáneas correspondientes y coloque los resultados en los parámetros de salida  $x$  y  $y$ .

### ★ ★ ★ EJERCICIOS ESTIMULANTES

- 10 Escríbase un procedimiento en Pascal llamado *MAX* con dos parámetros de valor enteros  $A$  y  $B$  y un parámetro variable entero  $C$ . El procedimiento debe utilizar exclusivamente las características del lenguaje Pascal que ya se estudiaron y devolver mediante el parámetro  $C$  el mayor de los parámetros  $A$  y  $B$ , que deben ser enteros positivos mayores de cero. (Sugerencia: considérese el uso de las funciones *trunc* y *round*.)

## PROBLEMAS DEL CAPÍTULO 4 PARA RESOLUCIÓN EN COMPUTADORA

### ★ PROBLEMAS ESENCIALES

- 1 Escríbase un procedimiento en Pascal para determinar el área de una figura de cuatro lados, dadas las coordenadas de los vértices de la figura. Utilícese el procedimiento de área de triángulos desarrollado en el capítulo.
- 2 Un año bisiesto es todo aquel divisible entre cuatro, excepto si es divisible entre cien, en cuyo caso será bisiesto solamente si es divisible entre 400. Por ejemplo, 1984, 2000, 2008 y 2400 son años bisiestos, pero 1700, 1961 y 2317 no lo son. Escríbase un procedimiento en Pascal llamado *bisiesto* cuyo pará-

metro de entrada sea un año y cuyo parámetro de salida sea un valor booleano que será *true* (verdadero) si el parámetro de entrada es año bisiesto.

- 3 Escribáse un procedimiento para calcular el valor del polinomio

$$ax^3 + bx^2 + cx + d,$$

donde  $a, b, c$  y  $d$  son parámetros de entrada enteros y  $x$  es un parámetro de entrada real. El valor del polinomio será el parámetro de salida. Después, escribáse un programa principal que utilice la salida para determinar el valor de

$$f[g(x)] \quad y \quad g[f(x)]$$

donde  $f(x) = ax^3 + bx^2 + cx + d$ , y  $g(x) = jx^3 + kx^2 + lx + m$ , donde  $a, b, c, d, j, k, l, m$  se obtienen de los datos de entrada.

## ★ ★ PROBLEMAS IMPORTANTES

- 4 Dado un camino circular de radio  $R$  para un automóvil deportivo, ¿cuál será la máxima velocidad posible? El coeficiente de adhesión (fricción) entre la superficie del camino y las ruedas del automóvil es aproximadamente 0.8 si las condiciones son buenas, y la fuerza requerida para mover al automóvil lateralmente está dada por el producto del peso del automóvil y el coeficiente de fricción. La fuerza centripeta que actúa sobre el automóvil es igual a  $m \cdot v^2/r$ , donde  $m$  es la masa del automóvil y  $v$  su velocidad. Escribanse un procedimiento y un programa completo en Pascal para determinar la velocidad máxima posible, dado un coeficiente de adhesión de 0.8 y siendo el peso del automóvil y el radio de la curva datos de entrada. El peso del automóvil es el producto de su masa y la aceleración de la gravedad (aproximadamente 9.81 metros por segundo al cuadrado en la superficie terrestre).
- 5 Dado un entero  $n$ , escribáse un programa en Pascal que utilice un procedimiento para dibujar un cubo cuyos lados estén formados por asteriscos, centrado en el dispositivo de salida. Supóngase un valor máximo de 20 para  $n$ . Por ejemplo, si el dato de entrada fue 5 y el dispositivo de salida tiene una anchura de 65 columnas, la salida se vería así:

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

Obsérvese que la figura no es cuadrada realmente, ya que el número de renglones por pulgada y el número de caracteres por pulgada no es el mismo en la mayor parte de los dispositivos de salida. Por ejemplo, muchas impresoras exhiben 10 caracteres por pulgada pero solamente 6 u 8 renglones por pul-

gada. Para hacer un poco más complicado el problema, considérese el problema de la perspectiva en la solución.

- 6 Los datos de entrada son cinco números reales que representan ángulos medidos en radianes. Constrúyase una tabla de valores de las funciones trigonométricas seno, coseno, tangente, cotangente, secante y cosecante del ángulo dado, con precisión de dos cifras decimales. Los resultados se deben exhibir como se muestra en el ejemplo.

Ejemplo de entrada:

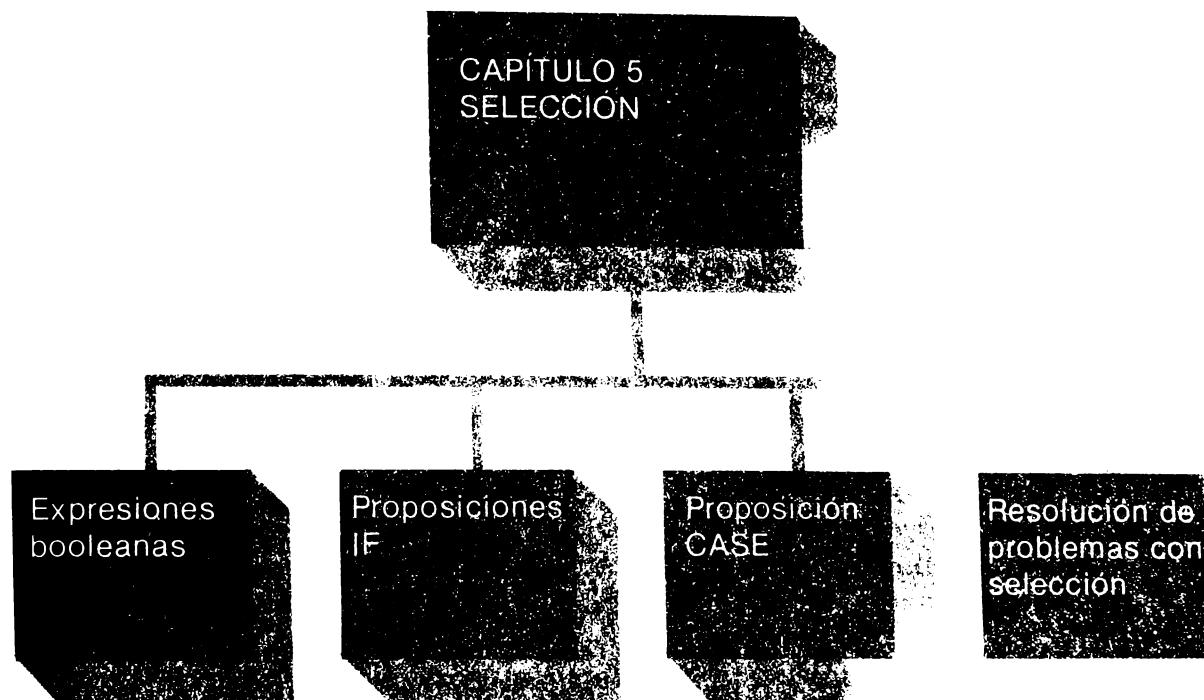
0.40      0.23      0.90      0.50      0.71

Ejemplo de salida

Ángulo	0.40	0.23	0.90	0.50	0.71
Seno	0.39	0.23	0.78	0.48	0.65
Coseno	0.92	0.97	0.62	0.88	0.76
Tangente	0.42	0.23	1.26	0.55	0.86
Cotangente	2.37	4.27	0.79	1.83	1.16
Secante	1.09	1.03	1.61	1.14	1.32
Cosecante	2.57	4.39	1.28	2.09	1.53



# CAPÍTULO 5



## SELECCIÓN

## OBJETIVOS

Después de completar este capítulo, el lector deberá ser capaz de:

- Evaluar expresiones booleanas que incluyan operadores booleanos y relacionales
- Reconocer y aplicar las proposiciones IF-THEN e IF-THEN-ELSE en problemas de decisión
- Reconocer y aplicar proposiciones IF anidadas
- Reconocer y aplicar la proposición CASE
- Resolver, probar y depurar problemas que incluyen selección

## PANORAMA GENERAL DEL CAPÍTULO

En muchos problemas es necesario tomar diferentes cursos de acción según si se cumple o no alguna condición. Por ejemplo, la conversión de la calificación numérica de un estudiante en un curso (en la escala de cero a 100) en una calificación de letra (como por ejemplo A, B, C, D o F) es un problema de este tipo. Se podría decidir que una calificación numérica de 95 debe convertirse en una calificación de letra A y una calificación numérica de 85 en B, etc., Los conceptos de programación presentados hasta ahora no incluyen un mecanismo para comparar valores y elegir distintos cursos de acción. En este capítulo se analizará y aplicará la toma de decisiones o selección y se introducirán las características del lenguaje Pascal que permiten emplear estos procesos en los programas.

Los programas en Pascal que se presentaron hasta este punto tienen una estructura común que se refleja en el orden de ejecución de las proposiciones. Dicho en forma más específica, las proposiciones de los programas se ejecutan en secuencia: se comienza por la primera proposición del programa principal y se continúa con las proposiciones que siguen en orden hasta ejecutarse la última (véase la Fig. 5-1).

Los lenguajes de alto nivel estructurados, como Pascal, incluyen proposiciones que pueden alterar el flujo de una secuencia de instrucciones. Técnicamente, estas proposiciones se conocen como **estructuras de control**. De hecho, la proposición de invocación de procedimientos que se introdujo en el capítulo anterior es un ejemplo de estructura de control. Cuando se invoca el procedimiento, el control se transfiere automáticamente del programa principal al procedimiento para regresar más tarde al programa principal. En este capítulo se estudiará una estructura de control fundamental llamada **selección**, mediante la cual se pueden ejecutar las proposiciones de manera condicional. Es decir, si se cumple una determinada condición, se ejecutará una secuencia de proposiciones; pero si la condición no se cumple, entonces se ejecutará una secuencia de proposiciones diferente. Por ejemplo, supóngase que se declara una variable llamada *núm* como entero y que contiene algún valor. Se desea determinar si *núm* contiene un número par o non y en se-

↓  
Proposición 1



↓  
Proposición 2



↓  
Proposición n



↓  
**FIN**

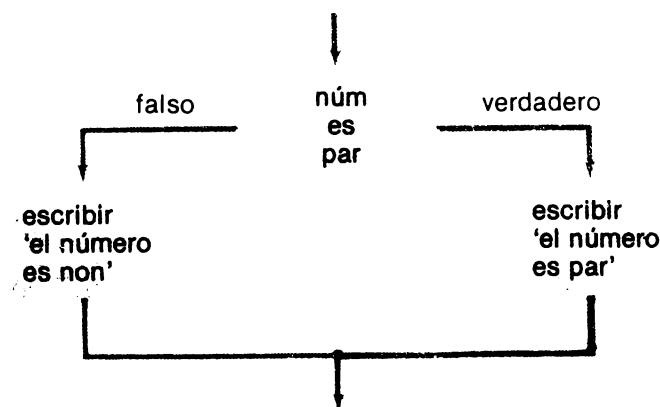
**Figura 5-1** Ejecución secuencial.

guida imprimir un mensaje que lo indique. La figura 5-2 representa el flujo de control de este proceso de decisión o selección.

Así, la condición que se va a probar (indicada por el rombo en la Fig. 5-2) es si *núm* contiene un número par. Si el contenido de *núm* contiene un número non, entonces la condición no se cumple y el control se transfiere a la proposición que exhibe el mensaje “El número es non.”

La proposición que se usa en Pascal (llamada proposición IF-THEN-ELSE) y que corresponde a esta estructura de control se muestra en seguida:

**Figura 5-2** Ejemplo de estructura de control de selección



```
IF núm MOD 2 = 0
THEN writeln ('El número es par.')
ELSE writeln ('El número es non.')
```

Recuérdese que MOD produce el residuo después de la división entera. En este caso el residuo puede ser cero o uno, lo cual indica que el número es par o non, respectivamente. En este capítulo se estudiarán en forma detallada las estructuras condicionales y la toma de decisiones mediante las proposiciones de selección de Pascal IF-THEN-ELSE, IF-THEN y CASE. En el siguiente capítulo se examinará otra estructura de control fundamental llamada *ciclo*. Los ciclos permiten la ejecución repetida de una secuencia de proposiciones mientras se cumple una condición.

## SECCIÓN 5.1 EXPRESIONES BOOLEANAS

Obsérvese que en el ejemplo anterior se probó la condición “núm MOD 2 = 0” para determinar la proposición que se debía ejecutar después. La condición “núm MOD 2 = 0” es una expresión que puede ser verdadera o falsa, según si núm contiene un número par o non. Las expresiones que pueden ser o bien falsas o bien verdaderas se llaman *expresiones booleanas* (por el matemático británico del siglo XIX George Boole). En esta sección se estudiarán en detalle las expresiones booleanas. La importancia de estas expresiones será evidente en la siguiente sección, donde se estudiará la proposición IF-THEN-ELSE.

### Variables booleanas

Una expresión booleanas en Pascal puede ser una variable booleana. A las variables booleanas solamente se les pueden asignar los valores *true* (verdadero) o *false* (falso). Considérese la siguiente declaración de una variable booleana llamada *conmuta*.

```
VAR conmuta : Boolean;
```

La siguiente proposición asigna el valor *true* a la variable booleana *conmuta*:

```
conmuta := true
```

La siguiente proposición asigna el valor *false* a *conmuta*:

```
conmuta := false
```

Es importante hacer notar que en Pascal estándar los valores de una variable booleana no se pueden obtener de los datos de entrada mediante la proposición *read*. (Recuérdese, empero, que sí se pueden exhibir mediante las proposiciones *write* y *writeln*.)

Las expresiones booleanas en Pascal pueden contener los siguientes operadores *relacionales*:

Pascal	Español
=	Igual a
<>	Diferente de
<=	Menor o igual que
>=	Mayor o igual que
>	Mayor que
<	Menor que

Supóngase que *núm1* y *núm2* se declaran como entero. Las siguientes son expresiones booleanas válidas en Pascal:

```
num1 > num2
num1 = num2
num1 <> num2
num2 <= num1
```

Dependiendo de cuáles sean los valores de *núm1* y *núm2*, cualquiera de estas expresiones booleanas puede ser falsa o verdadera. Por ejemplo, si *núm1* contiene al entero tres y *núm2* contiene al entero cinco, entonces la expresión booleana “*núm1 > núm2*” es falsa, ya que “3 > 5” es falso. La tabla que se muestra en seguida incluye más ejemplos con el valor de las expresiones booleanas para valores específicos de *núm1* y *núm2*.

<i>núm1</i>	<i>núm2</i>	Expresión booleana	valor
2	5	num1 < num2	true
0	1	num1 > num2	false
3	2	num1 > num2	true
8	7	num1 <> num2	true
7	7	num1 <= num2	true
7	7	num1 <> num2	false

Los operadores relacionales se pueden aplicar también a otros tipos de datos. Si se usan variables reales se obtienen resultados similares. Considerese ahora un ejemplo en el que *calif* se declara como variable de carácter en Pascal. Entonces la expresión “*calif > ‘A’*” es una expresión booleana válida. En este caso, el operador relacional *>* se refiere al ordenamiento de los caracteres en el sistema de cómputo, conocido como *ordenamiento lexicográfico* o *secuencia de ordenamiento*. Si el carácter almacenado en la variable *calif* aparece después del carácter ‘A’ en el ordenamiento del sistema de cómputo de que se trate, la expresión booleana se cumplirá; en caso contrario será falsa.

El ordenamiento de los caracteres puede variar en las diferentes versiones de Pascal y los distintos sistemas de cómputo, pero el Pascal estándar requiere que se cumplan ciertas relaciones en todos los casos. Estos requisitos son:

- Los valores de carácter que representan a los dígitos del cero al nueve deben quedar en el orden esperado, sin caracteres intermedios. Es decir, ‘0’ < ‘1’, ‘1’ < ‘2’,..., ‘8’ < ‘9’. Cabe hacer notar que aunque las relaciones son las mismas que en el caso de los enteros y los reales, éstos son caracteres, no números.
- Las letras mayúsculas de la *A* a la *Z* deben aparecer en el orden esperado (*‘A’* antes de *‘B’*, *‘B’* antes de *‘C’* y así sucesivamente), pero pueden existir caracteres intermedios. (Por ejemplo, algunos sistemas de cómputo IBM grandes pueden tener algunos caracteres entre los caracteres alfabéticos de mayúsculas.)
- Si se dispone de caracteres de minúsculas, deben obedecer la misma regla de ordenamiento que se requiere para los caracteres alfabéticos de mayúsculas (de modo que *‘a’* < *‘b’*, *‘b’* < *‘c’* y así sucesivamente). Por cierto, esta regla no requiere que los caracteres de minúsculas aparezcan antes de las letras mayúsculas. Así, no es posible dar por sentado, en general, que *‘a’* < *‘A’*.

Nótese que estas reglas implican que *‘A’* > *‘2’* pudiera ser verdadero o falso, dependiendo del ordenamiento lexicográfico particular que se utilice en el sistema de cómputo. Muchos sistemas de computadora emplean el conjunto de caracteres ASCII (*American Standard Code for Information Interchange*, código americano estándar para intercambio de información), que define el ordenamiento que se muestra en el apéndice F. Muchos sistemas IBM grandes utilizan el EBCDIC (*Extended Binary-Coded Decimal Interchange Code*, código de intercambio ampliado decimal codificado en binario) y se muestra también en el apéndice F.

Al usar los operadores relacionales, los valores que se comparan deben ser del mismo tipo de datos, con la excepción de que es posible comparar enteros con reales. Por ejemplo, las siguientes expresiones booleanas *no son válidas*, ya que los tipos de dato de los valores que se comparan son diferentes.

**5 > false      ‘B’ = 5.0      ‘0’ > = 0**

Las expresiones del tipo de datos real pueden provocar problemas cuando se usan con operadores relacionales. Por ejemplo, la expresión booleana “ $(1.0 / 3.0) * 3.0 = 1.0$ ” es al parecer verdadera, pero dado que el resultado de evaluar  $(1.0 / 3.0) * 3.0$  será probablemente  $0.99999\dots$  (dada la precisión limitada de la aritmética de reales en computadoras), es probable que la expresión booleana sea falsa.

Por tanto, en ocasiones será preciso excluir al tipo de datos real de las explicaciones. Los demás tipos de datos simples —entero, de carácter y booleana— se llaman *tipos de datos ordinales*. Estos tipos de datos se llaman ordinales porque los valores están ordenados y se pueden especificar mediante una lista. Por ejemplo, los enteros están ordenados y se puede preparar una lista que vaya desde *–máxint* hasta *máxint*. Como se ha visto, los caracteres también están ordenados y se pueden listar. Los valores booleanos *true* y *false* están ordenados ya que *false* < *true* en Pascal.

### Operadores booleanos

Las expresiones booleanas se pueden combinar para formar expresiones más complejas mediante los tres operadores booleanos (o lógicos) AND (“y”), OR

(“o”) y NOT (“no”). Por ejemplo, supóngase que se desea probar si el entero  $núm$  se encuentra entre uno y diez; es decir, si “ $1 < núm < 10$ ”. Ésta no es una expresión legal en Pascal; sin embargo, las dos expresiones booleanas “( $1 < núm$ )” y “( $núm < 10$ )” se pueden combinar mediante el operador booleano AND para formar la expresión booleana

$(1 < núm) \text{ AND } (núm < 10)$

Esta expresión será verdadera si tanto “( $1 < núm$ )” como “( $núm < 10$ )” se cumplen. En general, si  $P$  y  $Q$  representan expresiones booleanas, la expresión booleana “ $P \text{ AND } Q$ ” será verdadera únicamente cuando tanto  $P$  como  $Q$  sean verdaderas. En caso contrario, la expresión “ $P \text{ AND } Q$ ” será falsa. Se puede resumir el valor de la expresión “ $P \text{ AND } Q$ ” mediante la siguiente *tabla de verdad* que muestra el resultado de “ $P \text{ AND } Q$ ” para todos los valores posibles de  $P$  y  $Q$ .

$P$	$Q$	$P \text{ AND } Q$
falsa	falsa	falsa
falsa	verdadera	falsa
verdadera	falsa	falsa
verdadera	verdadera	verdadera

Por ejemplo, si  $P$  es falsa y  $Q$  verdadera, entonces, de acuerdo con el segundo renglón de la tabla, se ve que “ $P \text{ AND } Q$ ” es falsa. Supóngase que un programa en Pascal contiene la siguiente declaración:

**VAR** indic, comuta, prueba : Boolean;

Entonces el segmento de programa en Pascal

```
indic := 5 > 10;
comuta := 'A' < 'B';
prueba := indic AND comuta
```

producirá las asignaciones equivalentes

```
indic := false
comuta := true
prueba := false
```

cuando se ejecute el programa.

La expresión “ $P \text{ OR } Q$ ”, donde  $P$  y  $Q$  son expresiones booleanas, es verdadera cuando o bien  $P$  es verdadera o bien  $Q$  es verdadera, o bien tanto  $P$  como  $Q$  son verdaderas. La tabla de verdad que se muestra en seguida resume el empleo del operador OR.

<i>P</i>	<i>Q</i>	<i>P OR Q</i>
falsa	falsa	falsa
falsa	verdadera	verdadera
verdadera	falsa	verdadera
verdadera	verdadera	verdadera

Mediante esta tabla se puede ver que la expresión booleana

$$(5 > 10) \text{ OR } ('A' < 'B')$$

es verdadera, ya que “(*A*’ < ‘*B*’)” es verdadera.

Los operadores de suma, resta, multiplicación y división se llaman operadores *binarios*, ya que se aplican a dos operandos. De igual manera, los operadores booleanos AND y OR se consideran operadores binarios, puesto que también ellos se aplican a dos operandos.

Los operadores que se aplican a un solo operando se llaman operadores *unarios*. Por ejemplo, si *j* es una variable entera, entonces la expresión —*j* utiliza el operador unario —. El operador booleano NOT también es un operador unario que invierte el valor lógico de su operando. Por ejemplo, la expresión booleana “NOT (5 > 10)” es verdadera, ya que “(5 > 10)” es falsa. La siguiente tabla de verdad resume el efecto del operador NOT.

<i>P</i>	<i>NOT P</i>
falsa	verdadera
verdadera	falsa

La siguiente tabla ilustra todavía más las propiedades de NOT, AND y OR. Aquí, *núm* es una variable entera cuyo valor es 3 e *indic* es una variable booleana cuyo valor es *true*.

<i>Expresión Booleana</i>	<i>valor</i>
(1 > 0) AND (2 = 2)	true
NOT indic	false
(0 < 1) OR (0 > 1)	true
(5 <= 6) AND (2 > 3)	false
NOT (2 <> 2)	true
(num = 1) OR (5 >= 4)	true
NOT (num <= 3)	false

### Orden de los operadores

Recuérdese que los operadores de las expresiones aritméticas se aplican en un orden específico cuando la expresión contiene más de un operador. De manera similar, los operadores booleanos y los operadores relacionales tienen un orden de

$\text{NOT } p \text{ OR } q \text{ AND } r$

se evalúa por pasos como se muestra en seguida:

NOT p	(NOT tiene la prioridad más alta)
q AND r	(AND tiene la siguiente prioridad más alta)
NOT p OR q AND r	(OR tiene la prioridad más baja)

Por tanto, en las expresiones que emplean más de uno de los tres operadores booleanos, el orden de prioridad es primero NOT, después AND y por último OR. Como en el caso de las expresiones aritméticas, es posible usar paréntesis para forzar la evaluación de una expresión en cualquier orden deseado. Por ejemplo,

$\text{NOT } (p \text{ OR } q \text{ AND } r)$

se evaluaría mediante la aplicación en primer término del operador AND, después el operador OR y por último el operador NOT.

La siguiente tabla contiene más ejemplos. El orden de evaluación se muestra mediante paréntesis

expresión booleana	orden de evaluación
NOT p OR q	(NOT p) OR q
p OR q AND r	p OR (q AND r)
NOT p AND q OR NOT r	((NOT p) AND q) OR (NOT r)

En Pascal suelen surgir problemas cuando las expresiones booleanas incluyen tanto operadores relacionales ( $>$ ,  $<$ ; etc) como operadores booleanos (AND, OR y NOT). Los operadores relacionales tienen *más baja* prioridad que los operadores booleanos, por lo que una expresión como

$\text{NOT } 4 > 5$

es errónea, ya que se aplicaría primero el operador NOT al operando entero 4. El uso de paréntesis es necesario en este caso para obtener el resultado deseado

$\text{NOT } (4 > 5)$

lo cual producirá el valor *true*. Otro ejemplo está dado por la expresión

$1 < \text{núm} \text{ AND } \text{núm} < 10$

donde  $\text{núm}$  se declaró como variable entera. Puesto que la operación AND tiene más alta prioridad que el operador relacional  $<$ , tendría que evaluarse primero la expresión “ $\text{núm AND núm}$ ”. Naturalmente, esto producirá un error, ya que el

operador booleano AND requiere operandos booleanos. La forma correcta de esta expresión es

$$(1 < \text{núm}) \text{ AND } (\text{núm} < 10)$$

En el apéndice E se puede encontrar el orden de prioridad de todos los operadores. Cuando existan dudas acerca del orden de aplicación de los operadores en una expresión, utilíicense paréntesis para forzar el orden de evaluación deseado. Además de indicar el orden en que se deben evaluar las expresiones, los paréntesis pueden mejorar la legibilidad de los programas.

## EJERCICIOS DE LA SECCIÓN 5.1

**1** Determinese si las siguientes expresiones booleanas son verdaderas o falsas.

- (a)  $2 < 4$
- (b)  $-2 < 0$
- (c)  $0 > 1$
- (d)  $'5' < '6'$
- (e)  $'Z' < 'A'$
- (f)  $\text{true} < \text{false}$

**2** Determinese si las siguientes expresiones booleanas son verdaderas o falsas.

- (a)  $('b' < 'd') \text{ AND } (0 < 1)$
- (b)  $('c' >= 'f') \text{ OR } ('C' <= 'C')$
- (c)  $(5 < 1) \text{ OR } (0 > -1)$
- (d)  $\text{NOT } (2 = 2) \text{ AND } (\text{maxint} < \text{maxint})$

**3** Determinese el valor de las siguientes expresiones booleanas. Supóngase que  $p = \text{true}$ ,  $q = \text{false}$  y  $r = \text{true}$ .

- (a)  $\text{NOT } p \text{ OR } q \text{ AND } r$
- (b)  $\text{NOT } p \text{ AND } p$
- (c)  $q \text{ AND } p \text{ OR } \text{NOT } r$
- (d)  $\text{NOT } (p \text{ AND } q \text{ AND } r)$
- (e)  $\text{NOT } (p \text{ AND } \text{NOT } q \text{ AND } r)$

**4** Determinese el valor de las siguientes expresiones booleanas. Supóngase que  $p = \text{true}$ ,  $q = \text{true}$  y  $r = \text{false}$ .

- (a)  $(p \text{ AND } q) \text{ AND } (3 < 5)$
- (b)  $\text{NOT } (p \text{ AND } r) \text{ OR } (p \text{ OR } q)$
- (c)  $('0' < '2') \text{ AND } (p \text{ AND } r)$
- (d)  $\text{NOT } (p \text{ AND } r) \text{ OR } \text{NOT } (p \text{ AND } q)$

**5** Determinese el orden de aplicación de los operadores de las siguientes operaciones booleanas por medio de la inserción de paréntesis. Supóngase que *indic*, *conmuta* y *prueba* son variables booleanas.

- a)  $\text{NOT } \text{indic} \text{ OR } \text{NOT } \text{conmuta}$
- b)  $\text{indic} \text{ OR } \text{prueba} \text{ AND } \text{NOT } \text{conmuta}$
- c)  $\text{NOT } (\text{indic} \text{ AND } \text{conmuta} \text{ OR } \text{prueba})$
- d)  $\text{prueba} \text{ OR } \text{conmuta} \text{ AND } \text{indic}$

- 6 Conviértanse las siguientes expresiones en expresiones booleanas válidas mediante la inserción de paréntesis. Supóngase que *núm* es una variable entera.

- (a) NOT 1 < num
- (b) 2 > num OR num < -2
- (c) NOT 0 < num OR num > 10
- (d) 1 > num AND num > 0

## SECCIÓN 5.2 SELECCIÓN MEDIANTE LA PROPOSICIÓN IF

Ahora que se conocen los fundamentos de las expresiones booleanas, es posible continuar con las estructuras de control de selección que se emplean en Pascal. La primera estructura de control que se estudiará es la proposición IF-THEN-ELSE. Esta proposición tiene la siguiente estructura:

IF expresión booleana  
THEN proposición-1  
ELSE proposición-2

La proposición comienza con la palabra reservada IF seguida de una expresión booleana. Ésta va seguida de la palabra reservada THEN y una proposición en Pascal (o un grupo de proposiciones). Por último, se escribe la palabra reservada ELSE, seguida también de una proposición (o grupo de proposiciones) en Pascal. Obsérvese que no se escribe punto y coma inmediatamente antes del ELSE.

La proposición IF-THEN-ELSE se ejecuta de la siguiente manera. Se evalúa la expresión boolena, lo que produce un valor *true* (verdadero) o *false* (falso). Si la expresión es verdadera, se ejecutará la proposición 1 y se hará caso omiso de la proposición 2. Si la expresión es falsa, se pasará por alto la proposición 1 y se ejecutará la proposición 2.

Considérese el ejemplo anterior en que se trataba de determinar si el contenido de la variable entera *núm* era un número par o non. La proposición IF-THEN-ELSE

IF núm MOD 2 = 0  
THEN writeln ('El número es par.')  
ELSE writeln ('El número es non.')

hace que se evalúe la expresión booleana “núm MOD 2 = 0”. Si el numero es par, la expresión booleana será verdadera y se ejecutará la proposición “writeln ('El número es par.')”, haciéndose caso omiso de la proposición que sigue al ELSE. Si el número es non, la expresión booleana será falsa, y se ejecutará la proposición “writeln ('El número es non.')” y se hará caso omiso de la expresión que sigue al THEN.

En la figura 5-3 se muestra el diagrama de sintaxis que corresponde a la proposición IF-THEN-ELSE. Este diagrama también especifica la sintaxis de la proposición IF-THEN que se estudiará más adelante en esta sección.

Nótese que la proposición IF-THEN-ELSE es una sola proposición de Pascal, y debe estar separada de cualquier proposición que le siga mediante un signo de punto y coma.



**Figura 5-3** Diagrama de sintaxis de una proposición IF.

### Problema 5.1

*Dadas dos variables enteras núm1 y núm2, determine el valor más alto y exhibase.*

El problema se resuelve mediante la siguiente proposición IF-THEN-ELSE:

```

IF núm1 > núm2
THEN writeln ('El número más grande es ', núm1)
ELSE writeln ('El número más grande es ', núm2)
  
```

Supóngase que también se desea almacenar el valor más alto en la variable entera *máx*. Sería necesario escribir dos proposiciones tanto en la parte THEN como en la parte ELSE de la proposición. El Pascal exige que, si se coloca más de una proposición en la parte THEN o ELSE de una proposición IF-THEN-ELSE, deben estar agrupadas y delimitadas por las palabras BEGIN y END. La palabra reservada BEGIN se coloca antes de la primera proposición del grupo, y la palabra reservada END se coloca después de la última proposición del grupo. La solución del problema modificado tendrá este aspecto:

```

IF núm1 > núm2
THEN BEGIN
      writeln ('El número más grande es ', núm1);
      máx := núm1
    END
ELSE BEGIN
      writeln ('El número más grande es ', núm2);
      máx := núm2
    END;
  
```

Como siempre, las proposiciones individuales que se agrupan con las palabras BEGIN y END se separan con signos de punto y coma, pero no se utiliza punto y coma antes del ELSE.

Ahora se examinará la estructura general de una proposición IF-THEN-ELSE cuando va precedida de otra proposición y seguida de una proposición más, y cuando tiene además varias proposiciones en las partes THEN y ELSE. La estructura sería como sigue:

proposición; (\* antes de la proposición IF \*)

IF expresión booleana

THEN BEGIN

    proposición;

    proposición

.

.

    proposición

END

ELSE BEGIN

    proposición

    proposición

.

    proposición

END;

proposición (\* que sigue a la proposición IF \*)

He aquí un ejemplo más específico. *Cuenta* es una variable entera y *letra* una variable de carácter. Este segmento de programa se podría utilizar como parte de un ciclo (sin la proposición “*cuenta* := 0”) para determinar el número de caracteres que hay antes de ‘Z’ en el ordenamiento lexicográfico.

cuenta := 0; (\* inicializar cuenta en cero \*)

read (letra); (\* leer un carácter \*)

IF letra < ‘Z’

THEN BEGIN

        writeln (letra, ‘ aparece antes de Z.’);

        cuenta := cuenta + 1

    END

ELSE BEGIN

        writeln (letra, ‘no aparece antes de Z.’);

        writeln (‘Este carácter no se contará.’)

    END

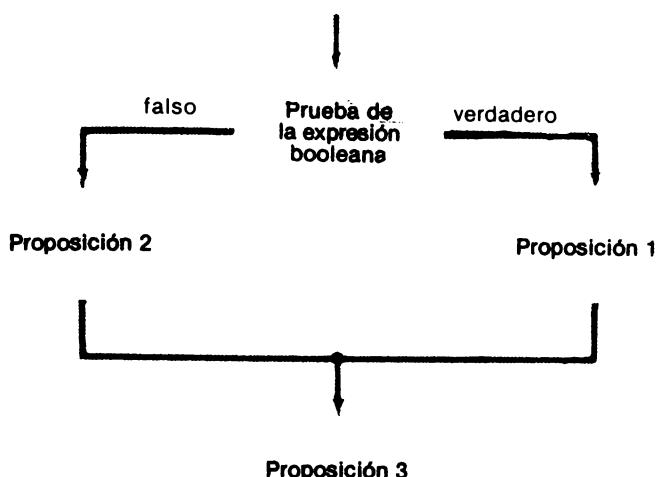
Tómese nota de que, cuando a una proposición IF-THEN-ELSE le sigue otra proposición, debe escribirse un signo de punto y coma para separar al IF-THEN-ELSE y a la proposición subsecuente.

La figura 5-4 ilustra la estructura de control de la proposición IF-THEN-ELSE. En esta figura, proposición 1 y proposición 2 pueden ser un grupo de proposiciones en Pascal, encerrado entre las palabras reservadas BEGIN y END, como debe ser.

### Proposición IF-THEN

Supóngase que se desea ejecutar una secuencia de proposiciones únicamente si se cumple cierta condición. En caso contrario, no se hará nada (y se continuará

IF expresión booleana  
THEN Proposición 1  
ELSE Proposición 2;  
Proposición 3



**Figura 5-4** Estructura de control IF-THEN-ELSE.

con la siguiente proposición). Por ejemplo, un programa para mantener los saldos de una cuenta de cheques puede incluir una proposición de selección que exhiba un aviso si la cuenta está sobregirada. En caso contrario, el programa continuará con la siguiente proposición. Supóngase que al estar sobregirada la cuenta, el saldo es negativo. La siguiente proposición IF-THEN (sin el ELSE) exhibiría el aviso:

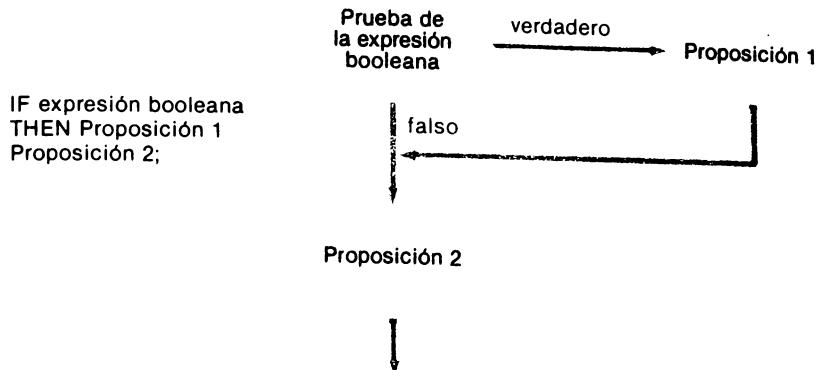
```
IF saldo < 0
THEN writeln ('Su cuenta está sobregirada.')
```

Ésta es una situación en la que se podría utilizar una proposición IF-THEN-ELSE, pero como la parte ELSE no se necesita, no deberá haber proposición alguna en la parte ELSE de la proposición IF-THEN-ELSE. Dado que esta situación se presenta con frecuencia en las soluciones de los problemas, el Pascal permite no sólo omitir la proposición que sigue al ELSE, sino también la palabra reservada misma.

Por ejemplo, supóngase que se desea exhibir un aviso si la calificación de un examen (representada por la variable entera *calificación*) queda fuera de la escala de cero a cien. La siguiente proposición IF-THEN logra ese objetivo.

```
IF (calificación < 0) OR (calificación > 100)
THEN writeln ('Esta calificación no es válida.')
```

En este caso el mensaje de error se exhibe únicamente si la calificación del examen es menor que cero o mayor que 100. Si la calificación es válida (es decir, está en la escala de cero a 100), la proposición IF-THEN no especifica acción alguna y la eje-



**Figura 5-5** Estructura de control IF-THEN.

cución continúa con la proposición que sigue al IF-THEN. La estructura general de la proposición IF-THEN con una proposición precedente y otra después es

proposición; (\* antes del IF-THEN \*)

IF expresión booleana

THEN proposición;

proposición (\* después del IF-THEN \*)

La figura 5-5 muestra la estructura de control de la proposición IF-THEN. Como se indicó antes, la proposición IF-THEN es una variante de la proposición IF-THEN-ELSE. De hecho, la proposición IF-THEN es totalmente equivalente a la siguiente proposición:

IF expresión booleana

THEN proposición

ELSE;

¡La “proposición” que sigue al ELSE es realmente una proposición de Pascal! Se llama **proposición vacía** y se incluye específicamente para utilizarse en situaciones de este tipo. Aunque en la mayor parte de los casos el uso de la proposición vacía es opcional, se verá que en algunas ocasiones es indispensable.

Algunos problemas se pueden resolver mediante una secuencia de proposiciones IF-THEN. Considérese, por ejemplo, el siguiente problema.

### Problema 5.2

*Determinese si la variable entera número es divisible entre dos, tres o cinco, y exhibase un mensaje apropiado.*

Este problema se resuelve fácilmente con tres proposiciones IF-THEN, cada una de las cuales emplea una expresión booleana en que se prueba si el número es divisible mediante el operador MOD. He aquí la solución:

```

IF número MOD 2 = 0
THEN writeln ('El número es divisible entre dos.');
IF número MOD 3 = 0
THEN writeln ('El número es divisible entre tres.');
IF número MOD 5 = 0
THEN writeln ('El número es divisible entre cinco.')

```

Si el valor de *número* es 4, entonces se exhibirá únicamente el primer mensaje. Sin embargo, si el valor de *número* es 30, aparecerán los tres mensajes en el dispositivo de salida.

### Proposiciones IF anidadas

Las proposiciones IF se pueden colocar dentro de otras proposiciones IF (es decir, *anidarse* en ellas). Por ejemplo, supóngase que se desea determinar cuál de dos enteros dados, *núm1* y *núm2*, es mayor. Puesto que es posible que los números sean iguales, se debe probar el cumplimiento de esta condición y exhibir un mensaje que exprese ese hecho. El siguiente segmento de programa en Pascal resuelve este problema:

```

IF núm1 >= núm2
THEN IF núm1 = núm2
    THEN writeln ('Ambos números son iguales a ', núm1)
    ELSE writeln ('El número más grande es ', núm1)
ELSE writeln ('El número más grande es ', núm2)

```

Este ejemplo utiliza una proposición IF anidada. Las proposiciones IF anidadas se pueden volver cada vez más complejas y difíciles de leer. Se ha incluido una sangría en la proposición IF anidada para que sea más fácil de leer y comprender. *Recuérdese que las sangrías al escribir proposiciones en Pascal no afectan la forma en que la computadora ejecuta esas proposiciones.* Por ejemplo, supóngase que se anida una proposición IF-THEN-ELSE dentro de una proposición IF-THEN y no se incluye sangría. Se podría escribir una proposición así en una sola línea, como se muestra en seguida:

```
IF a > b THEN IF a > c THEN máx := a ELSE máx := 0
```

Aquí se tienen dos proposiciones IF y un ELSE. ¿Cuál es la proposición IF que contiene al ELSE? Este problema se conoce en Pascal como *ELSE pendiente*. Para responder a la pregunta se hace notar la regla que un ELSE siempre corresponde al THEN precedente más cercano que no tenga ya un ELSE propio. Así, la proposición anterior, escrita con las sangrías adecuada, se convierte en

```

IF a > b
THEN IF a > c
    THEN máx := a
    ELSE máx := 0

```

Como se puede ver ahora, efectivamente se tiene una proposición IF-THEN-ELSE anidada dentro de una proposición IF-THEN. Por supuesto, las sangrías no tienen que ver con la forma como se ejecuta realmente la proposición. En Pascal, el ELSE se asocia siempre con el THEN más cercano que no tenga un ELSE propio. Para hacer hincapié en esto, considérese la siguiente forma de escribir la proposición:

```
IF a > b
THEN IF a > c
    THEN máx := a
ELSE máx := 0
```

Esto hace parecer en forma engañosa que el ELSE pertenece a la proposición IF externa. Es decir, parece que se fuera a ejecutar la proposición “máx := 0” solamente si la expresión booleana  $a > b$  fuera falsa. Sin embargo, esto es incorrecto, ya que el ELSE está asociado con el THEN más cercano, la sangría engañosa sugiere una interpretación diferente (e incorrecta). La mejor solución para evitar confusiones es sencilla: utilizar las sangrías para sugerir el orden *correcto* de evaluación de las proposiciones de un programa en Pascal.

Pero esto da pie a otra pregunta: ¿cómo se escribe la proposición si realmente se desea un IF-THEN-ELSE con un IF-THEN adentro? Esto se puede lograr si se encierra la proposición IF-THEN dentro de una pareja BEGIN-END. Esto *aisla* en forma efectiva a la proposición IF-THEN del proceso de asociación que se emplea para ligar el ELSE con el THEN más cercano. Esta solución se codifica así:

```
IF a > b
THEN BEGIN
    IF a > c
        THEN máx := a
    END
ELSE máx := 0
```

Considérese ahora el siguiente problema en el que se usa una proposición IF-THEN-ELSE anidada para encontrar el mayor de tres números.

### Problema 5.3

*Dadas tres variables enteras num1, num2 y num3, encuéntrese el mayor de estos tres números y almacénese en una variable entera llamada máx.*

Para resolver este problema se necesitan dos proposiciones IF-THEN-ELSE anidadas dentro de otra proposición IF-THEN-ELSE. Puesto que cada una de las proposiciones tiene su propio ELSE, no se requiere poner atención especial para forzar la asociación correcta de los componentes ELSE. No obstante, se seguirán los preceptos de sangría recomendados. He aquí la solución:

```
IF num1 > num2
THEN IF num1 > num3
```

```

THEN max := num1
ELSE max := num3
ELSE IF num2 > num3
    THEN max := num2
    ELSE max := num3

```

Se puede aprender mucho si se estudia con cuidado esta solución, ejecutándola varias veces con diferentes valores en *núm1*, *núm2* y *núm3*. Por ejemplo, como ejercicio, pruébese la solución con los valores *núm1* = 8, *núm2* = 10 y *núm3* = 5. ¿Qué sucederá cuando los tres números sean iguales?

En el siguiente problema se ilustra una solución del problema de convertir una calificación numérica en la correspondiente calificación de letra.

#### Problema 5.4

*Obtégase una calificación de examen entera de los datos de entrada y verifíquese que esté dentro de la escala de cero a 100. Si es así, exhibase la calificación de letra correspondiente empleando la escala de calificación 90 a 100 = A, 80 a 89 = B, 70 a 79 = C, 60 a 69 = D y 0 a 59 = F. Si la calificación de examen queda fuera de la escala permitida, exhibase un mensaje de error apropiado.*

La solución a este problema requiere varias proposiciones IF anidadas, cada una de las cuales procesa una de las escalas en las que puede quedar la calificación del examen. Una vez más, es conveniente como ejercicio probar a mano la solución con varios valores de entrada.

```

read (calificación);
IF (calificación < 0) OR (calificación > 100)
THEN writeln ('Calificación no válida: ', calificación)
ELSE IF calificación > = 90
    THEN writeln ('La calificación es A')
    ELSE IF calificación > = 80
        THEN writeln ('La calificación es B')
        ELSE IF calificación > = 70
            THEN writeln ('La calificación es C')
            ELSE IF calificación > = 60
                THEN writeln ('La calificación es D')
                ELSE writeln ('La calificación es F')

```

### EJERCICIOS DE LA SECCIÓN 5.2

- 1 Considérese la siguiente proposición:

```

IF (calif > = 90) OR (calif < 60)
THEN write ('Extrema')
ELSE write ('Media')

```

- a) ¿Qué se exhibirá si *calif* es igual a 90?  
b) ¿Qué se exhibirá si *calif* es cero?  
c) ¿Qué se exhibirá si *calif* es igual a 70?

2 Considérese la siguiente proposición IF anidada:

IF *a* > *b*  
THEN IF *a* > *c*  
    THEN write ('A es el más grande')

Escríbase una sola proposición IF que sea equivalente a éste pero que no esté anidada.

3 Considérese el siguiente segmento de programa:

```
x := 7;  
y := 8;  
IF x > y  
THEN x := x + 1  
ELSE y := y + 1
```

Determine el valor de cada una de las variables después de la ejecución.

4 Considérese el siguiente segmento de programa:

```
a := 5;  
b := 4;  
IF a > b THEN  
c := 999;  
d := 999;
```

Después de ejecutarse este segmento, ¿cuál de las siguientes afirmaciones es cierta?

- a) Ni *c* ni *d* quedan definidas.  
b) La variable *c* = 999 y *d* no está definida.  
c) La variable *d* = 999 y *c* no está definida.  
d) *c* = 999 y también *d* = 999.

5 Considérese el siguiente segmento de programa:

```
x := 10;  
y := 11;  
z := 12;  
IF (x > y) OR (z > y)  
THEN IF x > z  
    THEN IF y > z  
        THEN writeln ('Terminé.')
```

ELSE writeln ('No termino aún.')  
 ELSE writeln ('Nunca llega aquí.')

Determine si se exhibe alguno de los mensajes cuando se ejecuta el segmento, y cuál es.

- 6 Considérese el siguiente segmento de programa; se han declarado  $x$ ,  $y$  y  $z$  como variables enteras:

```
x := 1;
y := 2;
z := 3;
IF x > y
THEN IF y > z
  THEN IF x > z
    THEN writeln (x)
    ELSE writeln (y)
  ELSE writeln (z)
```

Determine qué es lo que se exhibe cuando se ejecuta el segmento.

- 7 ¿Qué se exhibe cuando se ejecuta el siguiente segmento de código?



```
a := 0;
b := -1;
IF a > 0
THEN writeln ('A')
ELSE IF b < 0
  THEN writeln ('B')
  ELSE writeln ('C')
```

### SECCIÓN 5.3 SELECCIÓN MEDIANTE LA PROPOSICIÓN CASE

Suponga el lector que está resolviendo un problema que requiere una selección entre varias alternativas. Esta selección se podría realizar usando muchas proposiciones IF anidadas, pero el Pascal cuenta con un mecanismo más apropiado: la proposición CASE.

La proposición CASE en Pascal es una estructura de control que permite la elección de un curso de acción de entre una lista de varias opciones. La proposición IF-THEN-ELSE permite la selección entre dos alternativas, pero CASE permite más de dos. Por ejemplo, considérese el problema de convertir una calificación de letra A, B, C, D o F en su equivalente en puntos de calificación numérica, 4.0, 3.0, 2.0, 1.0 ó 0.0, respectivamente. Supóngase que se declara la variable *califletra* como de carácter y *puntos* como real. La siguiente proposición IF de Pascal convierte la calificación de letra en su equivalente en puntos de calificación numérica.

```

IF califletra = 'A'
THEN puntos := 4.0
ELSE IF califletra = 'B'
    THEN puntos := 3.0
    ELSE IF califletra = 'C'
        THEN puntos := 2.0
        ELSE IF califletra = 'D'
            THEN puntos := 1.0
            ELSE puntos := 0.0

```

Aunque este código es correcto, se puede escribir en forma más concisa si se usa la proposición CASE. He aquí la proposición equivalente:

```

CASE califletra OF
    'A' : puntos := 4.0;
    'B' : puntos := 3.0;
    'C' : puntos := 2.0;
    'D' : puntos := 1.0;
    'F' : puntos := 0.0

```

END

La palabra reservada CASE va seguida de una expresión llamada *selector*. El selector solamente puede tener un valor que sea de un tipo de datos ordinal (es decir, no puede ser real). El valor del selector determina la proposición que se va a ejecutar. Después de la palabra reservada OF sigue una lista de proposiciones, cada una de las cuales está etiquetada con una constante del mismo tipo de datos que la expresión del selector. La proposición CASE termina con la palabra reservada END.

La proposición CASE anterior se ejecuta de la siguiente manera. Primero se determina el valor de *califletra* y se compara con la lista de constantes que se empleó para etiquetar las proposiciones del cuerpo de la proposición CASE. Cuando alguna de ellas concuerda con el valor, se ejecuta la proposición correspondiente.

La proposición que sigue a cada una de las etiquetas constantes puede ser una sola proposición en Pascal o cualquier grupo de proposiciones en Pascal delimitado por una pareja BEGIN-END. Obsérvese además que la proposición CASE en sí *no* requiere un BEGIN que corresponda al END del CASE.

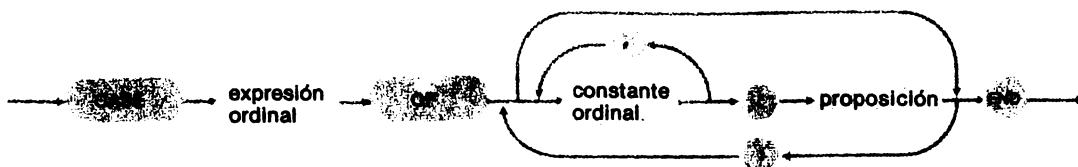
Nótese que las etiquetas constantes se separan de las proposiciones mediante signos de dos puntos. El diagrama de sintaxis de la proposición CASE se muestra en la figura 5-6. El flujo de control de la proposición CASE se ilustra en la figura 5-7.

Obsérvese que la proposición IF-THEN-ELSE no es más que un caso especial de la proposición CASE. Puesto que el tipo de datos booleano es ordinal, se puede parafrasear la proposición IF-THEN-ELSE general

```

IF expresión booleana
THEN proposición-1
ELSE proposición-2

```

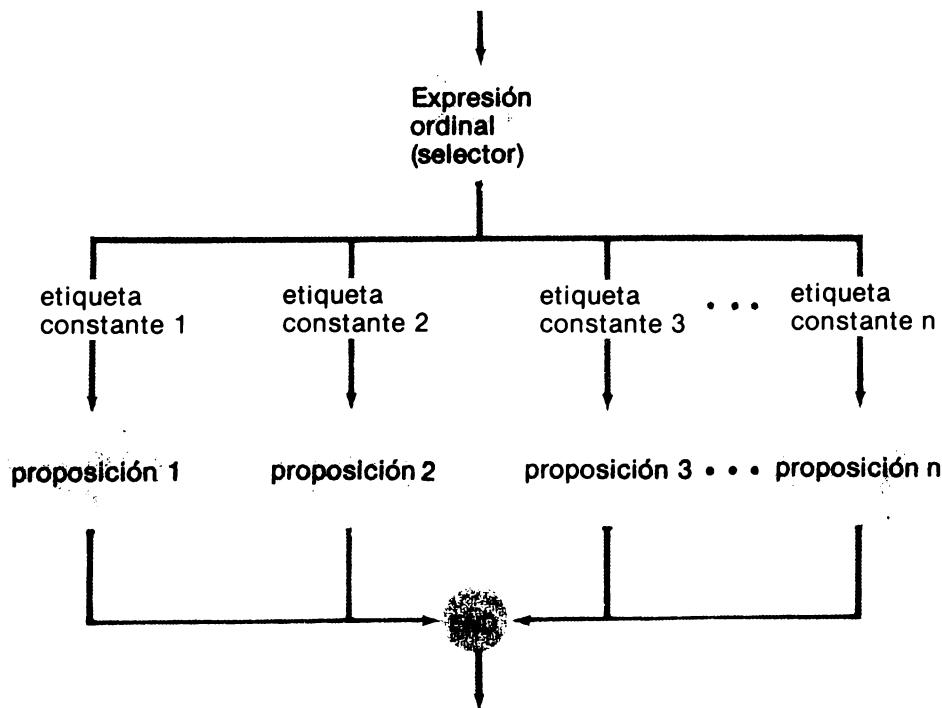
**Figura 5-6** Diagrama de sintaxis de la proposición CASE.

como la siguiente proposición CASE:

```

CASE expresión booleana OF
  true : proposición-1;
  false : proposición-2
END
  
```

¿Qué sucede si el valor de la expresión ordinal (selector) no concuerda con una de las etiquetas constantes? El resultado no se puede predecir, esto es, en algunos sistemas de Pascal se presentará un error durante la ejecución del programa, mientras que en otros sistemas se podrían pasar por alto todas las proposiciones del cuerpo de la proposición CASE sin dar al usuario indicación alguna de que se hizo esto. Cabe hacer notar que algunas versiones de Pascal cuentan con una etiqueta no estándar llamada *otherwise* (de otro modo) para una proposición que se ejecutará en caso de que ninguna de las demás etiquetas concuerde con el valor del selector. Considérese el siguiente código en Pascal que determina si una variable entera llamada *número* se encuentra entre uno y 10 (inclusive) y después, mediante una proposición CASE, especifica si el número es par o non:

**Figura 5-7** Estructura de control CASE.

2, 4, 6, 8, 10 : writeln ('El número es par.');

1, 3, 5, 7, 9 : writeln ('El número es non.')

END

Obsérvese que si el selector (*número*) es 2, 4, 6, 8 ó 10, entonces se ejecutará la proposición que identifica al número como par; si el número es 1, 3, 5, 7 ó 9 se ejecutará la otra proposición *writeln*. Si el número no está en la escala de uno a 10, la proposición CASE no se ejecuta siquiera. Esta proposición CASE ilustra la forma como se pueden combinar varias constantes con comas para etiquetar la misma proposición.

He aquí algunos puntos importantes referentes a la proposición CASE:

- Si dos o más constantes hacen referencia a la misma proposición, se deben separar las constantes con comas.
- Una etiqueta constante dada no puede aparecer dos veces o corresponder a dos o más proposiciones diferentes.
- El orden de las etiquetas constantes es arbitrario.
- Todos los valores que puedan resultar de la evaluación del selector se deben especificar en la lista de etiquetas constantes. Si no se requiere acción alguna para un valor determinado del selector, entonces ese valor debe aparecer como etiqueta constante de una proposición vacía.

Para ilustrar algunos de estos puntos, considérese la siguiente proposición CASE:

### CASE califletra OF

'A', 'B' : writeln ('Felicitaciones!');

'C' : writeln ('Trabajo satisfactorio.');

'D' : writeln ('Demasiada televisión.');

'E' : ; (\* No existe esta calificación \*)

'F' : writeln ('Malas noticias, inténtalo más adelante.');

END

Si *califletra* tiene el valor 'E', se ejecuta la proposición vacía y la computadora simplemente continúa con la proposición que sigue al CASE. De hecho, se pasa por alto la proposición CASE.

El siguiente problema se ocupa de determinar el número de días en un mes dado del año. La proposición CASE es ideal para este tipo de problema.

### Problema 5.5

*Los datos de entrada incluyen un entero que supuestamente está en la escala de uno a 12. Léase este entero y verifíquese que esté en la escala apropiada y, de no ser así, exhibase un mensaje adecuado. En caso contrario, utilícese el entero co-*

mo número de un mes (por ejemplo, 1 = enero, 2 = febrero, así sucesivamente). Exhibase el número de días del mes correspondiente. Si el dato de entrada es 2 (febrero), pidase al usuario que indique si el año actual es bisiesto o no.

El siguiente programa representa una solución en Pascal de este problema. Obsérvese que en la proposición CASE la etiqueta constante 2 corresponde a una proposición que contiene más de una proposición, por lo que se requiere una pareja BEGIN-END para delimitar las proposiciones. De hecho, este grupo de proposiciones se podía haber escrito en forma de un procedimiento y en ese caso hubiera bastado con una sola proposición (o sea la invocación del procedimiento) con lo que BEGIN y END no serían ya necesarios.

```

PROGRAM días (input, output);
(* Programa para determinar el número de días en un mes dado. *)
(* El mes debe estar en la escala de uno a 12, inclusive.      *)
VAR
    mes : integer;      (* número del mes          *)
    bisiesto : char;    (* 'S' o 's' si es año bisiesto *)
BEGIN
    write (Por favor escriba el número del mes (1 a 12): ');
    readln (mes);
    (* Validación del número de mes *)
    IF (mes < 1) OR (mes > 12)
        THEN writeln ('Disculpe, pero ese número de mes no es válido.')
        ELSE CASE mes OF
            1, 3, 5, 7, 8, 10, 12 : writeln ('31 días');
            4, 6, 9, 11 : writeln ('30 días');
            2 : BEGIN
                writeln ('Es bisiesto este año (S o N)? ');
                readln (bisiesto);
                IF (bisiesto = 'S') OR (bisiesto = 's')
                    THEN writeln ('29 días')
                    ELSE writeln ('28 días')
                END
            END
        END
    END.

```

En este programa se permite al usuario contestar afirmativamente a la pregunta sobre año bisiesto mediante el tecleo de una s mayúscula o minúscula. Obsérvese la serie de palabras END que se requiere y el uso de sangrías para sugerir la correspondencia de cada END con el CASE o BEGIN apropiado. La tabla que sigue ilustra la salida que se obtendría si se proporcionaran distintos datos al programa.

<i>datos de entrada</i>	<i>salida</i>
1	31 días
2S	29 días
2N	28 días
4	30 días
13	/ Discúlpeme, pero ese número de mes no es válido.

## EJERCICIOS DE LA SECCIÓN 5.3

- 1 Determiníse el valor que se exhibirá después de la ejecución de la siguiente proposición CASE:

```

letra := 'E';
CASE letra OF
  'A': writeln ('El valor es 1.');
  'E': writeln ('El valor es 5.');
  'I': writeln ('El valor es 9.');
  'O': writeln ('El valor es 15.');
  'U': writeln ('El valor es 21.');
END

```

- 2 Escríbase una proposición CASE equivalente a la siguiente proposición IF:

```

IF k = 0
THEN r := r + 1
ELSE IF k = 1
  THEN s := s + 1
  ELSE IF (k = 2) OR (k = 3) OR (k = 4)
    THEN t := t + 2

```

- 3 Considerése el siguiente código:

```

CASE i OF
  1:  a := a + 1;
  2:  b := b + 1;
  3, 4: c := c + 1

```

¿Cuáles de las siguientes afirmaciones sobre el código son ciertas?

- a) La proposición CASE se compilará sin error.
- b) La proposición CASE debe terminar con un END.
- c) La proposición CASE debe incluir una pareja BEGIN-END.
- d) El símbolo *i* puede ser una variable real.

4 Escribase una proposición CASE equivalente a la siguiente proposición IF.

```

IF (calif = 'D') OR (calif = 'F')
THEN writeln ('Trabajo deficiente.')
ELSE IF (calif = 'C') OR (calif = 'B')
THEN writeln ('Buen trabajo.')
ELSE IF calif = 'A'
THEN writeln ('Trabajo excelente.')

```

## SECCIÓN 5.4 RESOLUCIÓN DE PROBLEMAS CON SELECCIÓN

Considérese el siguiente problema sobre el calendario para resolverse en computadora, basado en algunos de los problemas que se han analizado antes: dado el mes, día y año escritos en forma numérica, exhibir el nombre del mes, el día del mes, el año y el día del año. Por ejemplo, si los datos de entrada son "3 15 1987", entonces la salida deberá ser "Marzo 15, 1987 dia 74". Si los datos de entrada corresponden a un año bisiesto, como por ejemplo "3 15 1988", entonces la salida correspondiente será "Marzo 15, 1988 dia 75".

Con el uso del diseño descendente, es posible dividir este problema en los siguientes subproblemas:

- SUBPROBLEMA 1: Obtener el mes, día del mes y el año de los datos de entrada.
- SUBPROBLEMA 2: Exhibir el nombre del mes.
- SUBPROBLEMA 3: Calcular el día del año.
- SUBPROBLEMA 4: Exhibir el día del mes, año y día del año.

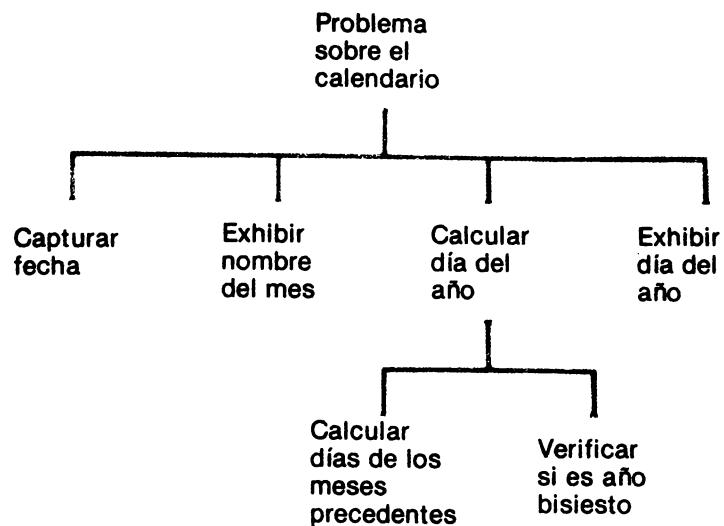
La figura 5-8 proporciona una representación gráfica de este diseño.

El único subproblema que requiere un análisis adicional es el cálculo del día del año. Una forma de determinar el día del año es calcular el número total de días en los meses anteriores al mes dado y después sumar el día del mes dado. Sin embargo, si el año es bisiesto y el mes es posterior a febrero, se deberá agregar un día adicional al total. El número de días en cada mes para un año no bisiesto está dado por la siguiente tabla:

<i>mes</i>	<i>Ene</i>	<i>Feb</i>	<i>Mar</i>	<i>Abr</i>	<i>May</i>	<i>Jun</i>	<i>Jul</i>	<i>Ago</i>	<i>Sep</i>	<i>Oct</i>	<i>Nov</i>	<i>Dic</i>
Días	31	28	31	30	31	30	31	31	30	31	30	31

El pseudocódigo del algoritmo deberá ser entonces:

- PASO 1 Obtener mes, día, año de los datos de entrada.
- PASO 2 Exhibir el nombre del mes.
- PASO 3 Calcular el día del año:  $\text{día del año} = (\text{total de días de los } i \text{ meses anteriores}) + \text{día del mes}$ . Si el año es bisiesto y mes  $> 2$ , sumar uno al día del año.
- PASO 4 Exhibir día, año y día del año.



**Figura 5-8** Diseño descendente del problema sobre el calendario.

Ya se puede escribir ahora el programa en Pascal. Se usa la proposición CASE para exhibir el nombre del mes, ya que existe una relación directa entre el número de mes y el nombre del mes. También se emplea una proposición CASE para determinar el día del año. Un procedimiento llamado *bisiesto* determina si el año es bisiesto o no. En el programa en Pascal que sigue se omitieron a propósito las proposiciones que requiere el procedimiento *bisiesto*. Especificarlas constituye un buen ejercicio para el estudiante.

```

PROGRAM calendario (input, output);
(* Convertir la representación numérica de mes, día y año *)
(* en nombre del mes, día, año y día del año. *)
VAR
  mes, día anio, díanual : integer;
  bis : Boolean;
PROCEDURE bisiesto (anio : integer; VAR bis : Boolean);
(* Procedimiento para determinar si "anio" es un año bisiesto. *)
(* Si así fuera, se asignará true a la variable "bis", *)
(* en caso contrario se le asignará false. *)
BEGIN
  (* Terminar este procedimiento es *)
  (* tema del ejercicio 4. *)
END;
(* Sigue el programa principal ... *)
BEGIN
  (* Obtener mes, día y año de los datos de entrada *)
  readln (mes, día, anio);
  (* Validar el número de mes y exhibir el nombre *)
  IF (mes < 1) OR (mes > 12)

```

THEN write ('INCORRECTO')  
ELSE BEGIN

```

CASE mes OF
  1 : write ('Enero ');
  2 : write ('Febrero ');
  3 : write ('Marzo');
  .
  4 : write ('Abril ');
  5 : write ('Mayo ');
  6 : write ('Junio ');
  7 : write ('Julio ');
  8 : write ('Agosto ');
  9 : write ('Septiembre ');
  10 : write ('Octubre ');
  11 : write ('Noviembre ');
  12 : write ('Diciembre ')

END;
(* Calcular día del año *)
CASE mes OF
  1 : dianual := día;
  2 : dianual := 31 + día
  3 : dianual := 31 + 28 + día;
  4 : dianual := 31 + 28 + 31 + día;
  5 : dianual := 31 + 28 + 31 + 30 + día;
  6 : dianual := 31 + 28 + 31 + 30 + 31 + día;
  7 : dianual := 31 + 28 + 31 + 30 + 31 + 30 +
    día
  8 : dianual := 31 + 28 + 31 + 30 + 31 + 30 +
    31 + día;
  9 : dianual := 31 + 28 + 31 + 30 + 31 + 30 +
    31 + 31 día;
  10 : dianual := 31 + 28 + 31 + 30 + 31 + 30 +
    31 + 31 + 30 + día;
  11 : dianual := 31 + 28 + 31 + 30 + 31 + 30 +
    31 + 31 + 30 + 31 + día;
  12 : dianual := 31 + 28 + 31 + 30 + 31 + 30 +
    31 + 31 + 30 + 31 + 30 + día

END;
(* Verificar si es año bisiesto *)
bisiesto (anio, bis);
IF bis AND (mes > 2)
THEN dianual := dianual + 1;
(* Exhibir día, año y día del año *)
write (día:1, ',', anio:1, ' Día ', dianual:1)
END
END.
```

---

---

 3 5 1990
 

---

 Marzo 15, 1990 día 74
 

---

*Programa calendario: ejemplo de ejecución*

Obsérvese que en el programa *calendario* el programa principal no es en esencia más que una proposición IF-THEN-ELSE. De hecho, la mayor parte del programa está contenida en la parte ELSE de la proposición. Obsérvese además que debe estar delimitada por la pareja BEGIN-END, ya que hay más de una proposición. En realidad, las proposiciones CASE que corresponden a la misma etiqueta constante se pueden combinar en una sola proposición CASE en la que ambas proposiciones estén delimitadas por una pareja BEGIN-END. Por último, el cálculo del día del año en la segunda proposición CASE es torpe y puede mejorarse. Como alternativa, se podrían calcular las sumas en cada proposición antes de escribir el programa. Entonces, por ejemplo, la proposición con la etiqueta 3 sería

3 : dianual := 59 + día;

y la proposición con la etiqueta 4 sería

4 : dianual := 90 + día;

y así sucesivamente. Otro método de calcular el día del año utiliza ciclos, que se estudiarán en el siguiente capítulo. Un método más para realizar este cálculo emplea la recursión, tema que se analizará en el capítulo 7.

## SECCIÓN 5.5 TÉCNICAS DE PRUEBA Y DEPURACIÓN

### Expresiones booleanas

El uso equivocado de expresiones booleanas en un programa en Pascal puede ser causa de muchos errores. Por ejemplo, la expresión booleana

**NOT (P AND Q)**

no es igual a la expresión booleana

**NOT P AND NOT Q.**

Por ejemplo, supóngase que  $P$  es la expresión booleana “ $núm = 0$ ” y  $Q$  es la expresión booleana “ $suma = 1$ ”. Decir que no ambas  $P$  y  $Q$  son verdaderas es lógicamente lo mismo que decir que o bien no  $P$  es verdadera o no  $Q$  es verdadera.

Es decir, o bien  $núm \neq 0$  o bien  $suma \neq 1$ . Así, las relaciones

$\text{NOT } (\text{P AND Q})$       y       $\text{NOT } ((\text{núm} := 0) \text{ AND } (\text{suma} = 1))$

son lógicamente iguales a

$(\text{NOT P}) \text{ OR } (\text{NOT Q})$       y       $(\text{núm} < > 0) \text{ OR } (\text{suma} < > 1)$ .

De manera similar,

$\text{NOT } (\text{P OR Q})$       y       $\text{NOT } ((\text{núm} := 0) \text{ OR } (\text{suma} = 1))$

son lógicamente iguales a

$(\text{NOT P}) \text{ AND } (\text{NOT Q})$       y       $(\text{núm} < > 0) \text{ AND } (\text{suma} < > 1)$ .

Estas relaciones lógicas se conocen como *leyes de DeMorgan*.

## Paréntesis

Es conveniente usar paréntesis en las expresiones booleanas que contiene tanto operadores booleanos como operadores relacionales o aritméticos. Por ejemplo, supóngase que se desea probar si el valor en una variable de carácter llamada *letra* se encuentra entre 'A' y 'Z'. Se podría escribir la siguiente expresión booleana:

$(\text{letra} >= 'A') \text{ AND } (\text{letra} <= 'Z')$

Los paréntesis son necesarios en esta expresión, ya que su omisión provocaría la aplicación del operador AND a los operandos 'A' y *letra*, ninguno de los cuales es booleano.

Aun cuando no sean necesarios los paréntesis en una expresión, en ocasiones conviene incluirlos, ya que esto muchas veces mejora la legibilidad de las expresiones. Si el lector no está seguro del orden en que se realiza la aplicación de operadores, se puede forzar un orden determinado mediante los paréntesis. Por ejemplo, supóngase que las variables *número*, *carácter* y *lógico* se declaran como de tipo booleano. Entonces la expresión

$\text{NOT } \text{número} \text{ OR } \text{carácter} \text{ AND } \text{lógico}$

es más fácil de leer y comprender cuando se escribe así:

$(\text{NOT } \text{número}) \text{ OR } (\text{carácter} \text{ AND } \text{lógico})$ .

## Validación

Algunos errores se deben a valores no válidos en los datos de entrada. Para evitar estos errores se recomienda incluir en los programas una prueba para detectar da-

tos no válidos después de la captura de los valores. Por ejemplo, el segmento de programa que se muestra en seguida espera un número positivo llamado *depósito*, el cual indica el número de dólares depositado en una cuenta de cheques. Se utiliza una proposición IF para validar los datos de entrada.

```
read (depósito);
IF depósito < = 0
THEN writeln ('Datos no válidos.')
```

Cuando se emplean proposiciones CASE siempre es prudente validar la escala del valor que se emplea como selector. En Pascal estándar, si el valor del selector no concuerda con alguna de las etiquetas constantes, el resultado no está definido. Por ejemplo, supóngase que una variable de carácter llamada *código* puede adquirir los valores 'A', 'B', 'C', 'D', 'E', 'V', 'W', 'X', y 'Z'. En ese caso, la siguiente proposición IF valida el valor del selector, lo que da lugar a la ejecución de la proposición CASE o la proposición *writeln* que exhibe un mensaje de diagnóstico:

```
IF((código > = 'A') AND (código < = 'E') OR
   (código > = 'V') AND (código < = 'Z'))
THEN CASE código OF
      'A', 'B', 'C', 'D', 'E': writeln ('Código de primeras letras.');
      'V', 'W', 'X', 'Y', 'Z': writeln ('Código de últimas letras.')
END
ELSE writeln ('Código de letra no válido: código')
```

La siguiente lista incluye algunos recordatorios importantes para ayudar al lector en la prueba y depuración de sus programas en Pascal cuando se utiliza la selección.

## RECORDATORIOS DE PASCAL

- Al evaluarse, las expresiones booleanas producen únicamente los valores *true* y *false*.
- El operador AND produce el valor *true* sólo cuando son verdaderos ambos operandos; en caso contrario el valor es *false*.
- El operador OR produce el valor *true* cuando cualquiera de los operandos es verdadero o cuando ambos son verdaderos; sólo se obtiene el resultado *false* cuando ambos operandos son falsos.
- El operador NOT invierte el valor lógico del operando.
- Los operadores lógicos NOT, AND y OR se evalúan en ese orden y antes que cualquier operador aritmético o relacional de la misma expresión, a menos que se usen paréntesis para forzar un orden de evaluación diferente. Una forma fácil de recordar el orden de evaluación de estos operadores es tomar nota de que sus nombres no aparecen en orden alfabético:

<u>No</u>	<u>Aparecen en</u>	<u>Orden</u>
<u>Not</u>	<u>AND</u>	<u>Or</u>

- En algunos casos se deben usar paréntesis para evitar errores de sintaxis:  

$$(\text{núm} > 0) \text{ AND } (\text{núm} < 10)$$
- En una proposición IF-THEN-ELSE *no* debe aparecer un punto y coma antes del ELSE:

```
IF a > 0
THEN b := b + 1;      (* ¡ERROR! *)
ELSE a := a + 1
```

- Un signo de punto y coma inmediatamente después de THEN o ELSE implica que existe una proposición vacía antes del punto y coma.

```
IF a > b
THEN ;      (* proposición vacía *)
IF b < 0
THEN b := b + 1
ELSE ;      (* proposición vacía *)
```

- Se requiere una pareja BEGIN-END cuando se incluye más de una proposición en la parte THEN o en la parte ELSE de una proposición IF.
- En una proposición CASE el valor del selector debe ser de un tipo ordinal (no real).
- Al evaluarse, el selector debe producir uno de los valores especificados por las etiquetas constantes.
- Es una proposición IF anidada, el ELSE se asigna al THEN precedente más cercano que no tenga un ELSE correspondiente. Las sangrías pueden hacer más legibles las proposiciones anidadas.

```
IF núm = 0
THEN IF súm = 0
    THEN cuenta := cuenta + 1
    ELSE total := total + 1
ELSE núm := 2 * núm
```

## SECCIÓN 5.6 REPASO DEL CAPÍTULO

En este capítulo se analizó la toma de decisiones, o selección, en el lenguaje Pascal. Las proposiciones que pueden alterar el flujo de una secuencia de instrucciones se conocen como estructuras de control. Las estructuras de control IF-THEN e IF-THEN-ELSE emplean expresiones booleanas para determinar el flujo de control. La proposición IF-THEN selecciona una proposición adicional (o grupo de instrucciones delimitadas por una pareja BEGIN-END) que se debe ejecutar si la expresión booleana resulta verdadera. La proposición IF-THEN-ELSE selecciona una instrucción de una pareja de instrucciones (o de grupos de instrucciones) con base en el valor (*true* o *false*) de la expresión booleana. Las proposiciones IF de ambos tipos se pueden anidar de manera que una proposición IF puede contener otras instrucciones IF.

La proposición CASE en una estructura general de control en Pascal que permite seleccionar una de entre posiblemente muchas instrucciones alternativas con base en el valor de una expresión que produce un valor ordinal.

Las referencias que siguen resumen las funciones de selección con que cuenta el lenguaje Pascal.

## REFERENCIAS DE PASCAL

### 1 Expresiones booleanas

1.1 A las variables booleanas solamente se les pueden asignar los valores *true* y *false*:

**conmuta := false**

#### 1.2 Operadores relacionales

= < > <= >= < >

#### 1.3 Operadores booleanos:

**AND      OR      NOT**

1.4 Expresiones representativas (*indic* y *conmuta* se declaran como booleanas; *núm* se declara como entera):

**indica AND conmuta**

**NOT conmuta OR indic**

**(núm > 0) OR conmuta**

**(núm > 1) AND (núm < = 10)**

1.5 Orden de evaluación de los operadores cuando no hay paréntesis:

<b>NOT</b>					
<b>AND</b>	/	DIV	MOD	*	
<b>OR</b>	+	-			
=	<>	<=	>=	<	>

### 2 Proposición IF-THEN-ELSE

#### 2.1 Forma general:

**IF expresión booleana**

**THEN proposición-1**

**ELSE proposición-2**

Ejemplo específico:

**IF núm > 0**

**THEN writeln ('El número es positivo.')**

**ELSE writeln ('El número es negativo o cero.')**

## 2.2 Proposición IF-THEN:

IF expresión booleana  
THEN proposición

Ejemplo específico:

```
IF (calif >= 90) AND (calif <= 100)
THEN writeln ('Su calificación es A.')
```

## 2.3 Proposición vacía:

IF expresión booleana  
THEN proposición  
ELSE; (\* proposición vacía antes del punto y coma \*)  
IF expresión booleana  
THEN ; (\* proposición vacía antes del punto y coma \*)

## 2.4 Proposición IF anidada (proposición IF dentro de un IF):

```
IF calif1 < calif2
THEN IF calif1 < calif3
    THEN chica := calif1
    ELSE
ELSE IF calif2 < calif3
THEN chica := calif2
ELSE chica := calif3
```

## 3 Proposición CASE

Forma general:

```
CASE expresión ordinal OF
    etiqueta-constante-1 : proposición 1;
    etiqueta-constante-2 : proposición 2;
    ...
    etiqueta-constante-n : proposición n;
END
```

Ejemplo específico:

```
CASE califletra OF
    'A' : writeln ('Excelente.');
    'B' : writeln ('Muy bien.');
    'C' : writeln ('Bien.');
    'D' : writeln ('Regular.');
    'F' : writeln ('Deficiente.')
END
```

Hasta aquí se han estudiado tres estructuras de control: secuencia, procedimiento y selección. La secuencia se analizó en el capítulo 2 y consiste en la ejecución de las proposiciones en el orden en que están escritas, una después de otra. En el capítulo 4 se introdujeron los procedimientos. Éstos alteran el flujo del programa para ejecutar como una unidad un grupo de proposiciones al que se hace referencia por nombre y posiblemente con ayuda de una lista de parámetros verdaderos en una proposición. En este capítulo se analizó la toma de decisiones mediante la selección (expresiones booleanas y proposiciones IF, y expresiones ordinales y proposiciones CASE).

En el siguiente capítulo se introducirán los ciclos, otra estructura de control de gran importancia para preparar soluciones a los problemas. Los ciclos permiten la repetición controlada de una secuencia de proposiciones. Se verán varias formas de estructuras de control cíclicas.

Como descubrirá el lector, estas cuatro estructuras de control (secuencia, procedimiento, selección y ciclo) son suficientes para escribir programas estructurados en Pascal capaces de resolver casi cualquier problema al que pueda enfrentarse.

### Palabras clave del capítulo 5

estructura de control	proposición CASE
expresión booleana	proposición IF-THEN
operador AND	proposición IF-THEN-ELSE
operador NOT	proposición vacía
operador OR	proposiciones IF anidadas
operador binario	secuencia de ordenamiento
operador booleano	selección
operador unario	selector
operadores relacionales	tipos de datos ordinales
ordenamiento lexicográfico	variable booleana

## EJERCICIOS DEL CAPÍTULO 5

### ★ EJERCICIOS ESENCIALES

- 1 ¿Qué sucede en el sistema del lector cuando se ejecuta el siguiente segmento de código?

```
val := 3;
CASE val OF
  1: writeln ('uno');
  2: writeln ('dos')
END;
writeln ('Después del CASE')
```

Si el lector puede habilitar e inhabilitar la verificación diagnóstica, pruébese la ejecución del segmento de ambas maneras.

- 2 Las versiones no estándar de Pascal muchas veces incluyen un componente OTHERWISE o ELSE en las proposiciones CASE para incluir una proposición que se ejecuta únicamente si el valor del selector no concuerda con ninguna de las etiquetas constantes. Determinese si el compilador de Pascal que se emplea cuenta con ese componente y utilícese para manejar el problema del ejercicio 1.
- 3 Indíquese los pasos necesarios para la expresión

$$(a < b) < (c < d)$$

donde  $a$ ,  $b$ ,  $c$  y  $d$  son del mismo tipo. Evalúese la expresión para  $a$ ,  $b$ ,  $c$  y  $d$  iguales a 1, 2, 3 y 4, respectivamente, y para 4, 3, 2 y 1.

- 4 Determinese el número aproximado de dígitos decimales que se pueden almacenar en una variable real en el sistema que se utilice. Para ello determinese cuál es la primera de las siguientes expresiones que produce el valor *true*. Trátese de explicar la razón de que funcione este método.

1.01	=	1.0
1.001	=	1.0
1.0001	=	1.0
1.000001	=	1.0
y así sucesivamente		

- 5 A ciertos estudiantes se les dice que su calificación final será el promedio de las cuatro calificaciones más altas de entre las cinco que se han obtenido en el curso. Escríbase un procedimiento llamado *promclase* con cinco parámetros de entrada (las calificaciones obtenidas) y un parámetro de salida (la calificación promedio) que lleve a cabo este cálculo. Una variación un poco más difícil es resolver el mismo problema pero suponiendo que la calificación se basará en el promedio de las cuatro calificaciones más altas de entre seis calificaciones obtenidas.

### ★ ★ EJERCICIOS IMPORTANTES

- 6 Escríbase un procedimiento llamado *coma* que exhiba un valor entero de siete dígitos o menos, con comas en las posiciones usuales. Por ejemplo, el entero 1279621 se exhibiría como 1,279,621.
- 7 A partir del primer carácter, los datos de entrada deben incluir una constante *char* válida en Pascal. Escríbase un programa que determine si estos datos de entrada son válidos. (Recuérdese que la forma usual es un apóstrofo, el carácter y otro apóstrofo, sin olvidar el caso especial "'''".)

- 8 El valor de  $n!$  ( $n$  factorial) se puede definir en forma recursiva como

$n$	$n!$
0	1
$> 0$	$(n - 1)! \cdot n$

Escríbase un procedimiento en Pascal que determine el valor de  $n!$  dado un parámetro de entrada  $n$ . Si  $n < 0$ , exhibase un mensaje de error apropiado y prodúzcase el valor  $-1$ .

## PROBLEMAS DEL CAPÍTULO 5 PARA RESOLUCIÓN EN COMPUTADORA

### ★ PROBLEMAS ESENCIALES

- 1 El costo de enviar por correo una carta de primera clase es \$0.22 para cartas que pesan menos de una onza y \$0.22 más \$0.17 por onza adicional o fracción para cartas que pesan más de una onza. Dado el peso de una carta como un número real de onzas, exhibase el costo de enviar la carta en la forma que se muestra en el ejemplo.

Ejemplo de entrada:

1.01

Ejemplo de salida:

Peso	Costo
1.01	0.39

- 2 Léase un entero positivo  $n$  (de menos de cinco dígitos) y un entero positivo  $d$  (de exactamente un dígito). Si  $d$  aparece en la representación decimal de  $n$ , entonces exhibase el número  $n$  con un acento circunflejo (̄) debajo de cada aparición de  $d$ . En caso contrario, exhibase el mensaje “d no aparece en n”.

Ejemplo de entrada:

100 0

Ejemplo de salida:

100

Ejemplo de entrada:

152 3

Ejemplo de salida:

3 no aparece en 152.

- 3 Determínense las raíces de la ecuación cuadrática

$$ax^2 + bx + c = 0$$

donde  $a$ ,  $b$  y  $c$  se dan como números reales con valores inferiores a 100 en los datos de entrada. Exhibanse los valores de  $a$ ,  $b$  y  $c$  y las raíces en una de las siguientes formas:

$$a = xx.x \quad b = xx.x \quad c = xx.x$$

Dos raíces reales diferentes: xxx.xxx y xxx.xxx

o Dos raíces reales iguales: xxx.xxx

o Dos raíces imaginarias diferentes: xxx.xxx + xxx.xxx  $i$

El número de dígitos que se emplee para exhibir las raíces de la ecuación puede variar.

Ejemplos de entrada:

$$2.0 \quad 3.0 \quad 1.0$$

Ejemplos de salida:

$$a = 2.0 \quad b = 3.0 \quad c = 1.0$$

Dos raíces reales diferentes: -1.000 y -0.500

- 4 Los datos de entrada contienen un entero. Cuando este entero se escribe como número de cuatro dígitos, con ceros a la izquierda si es necesario, los dos dígitos de la extrema izquierda representan un valor de horas en la escala 00 a 23 y los dos dígitos de la extrema derecha representan un valor de minutos en la escala 00 a 59. Escríbase este entero en la forma hh:mm. Hágase que aparezcan los cuatro dígitos, aun cuando sean ceros.

Ejemplo de entrada:

402

Ejemplos de salida:

04:02

### ★ ★ PROBLEMAS IMPORTANTES

- 5 Dado un valor de entrada que representa una cantidad de dólares inferior a \$1000, exhibase la cantidad y su equivalente en forma de palabras:

Salida:

**Trescientos cincuenta y siete dólares con noventa y ocho centavos**

- 6** Conviértase un número positivo hexadecimal de cuatro dígitos en su equivalente en base 10. Un dígito hexadecimal es uno de los dígitos 0 a 9 o A (10), B (11), C (12), D (13), E (14) o F (15). El equivalente decimal de un número hexadecimal de la forma  $abcd$  es  $a \cdot 16^3 + b \cdot 16^2 + c \cdot 16 + d$ .
- 7** Los datos de entrada de este problema son una tabla de tasas de descuento en compras al mayoreo de un producto determinado (como datos de entrada en el formato que se muestra en el ejemplo), una tasa de impuesto específica para la compra, una tasa de descuento por pago en efectivo, el número de unidades del producto deseado y el costo unitario. Determinese el costo total. El primer carácter de los datos de entrada será *E* si el pago es en efectivo y *C* si la compra es a crédito. El resultado será el importe de los descuentos redondeado al centavo más cercano y el costo final de la compra. Si procede, se aplicará primero el descuento por compra al mayoreo y después el descuento por pago en efectivo. Exhíbanse los resultados con el formato que se muestra en seguida. La tabla de descuentos tendrá entre uno y cinco renglones.

Ejemplo de entrada:	(Comentarios, no son parte de los datos)
C	(pago en efectivo)
0.05	(5% de descuento por pago en efectivo)
712	(número de unidades adquiridas)
12.04	(cada unidad cuesta \$12.04)
5	(número de renglones de la tabla)
10 0.00	(de 1 a 10, no hay descuento)
25 0.02	(de 11 a 25, 2%)
100 0.04	(de 26 a 100, 4%)
500 0.07	(de 101 a 500, 7%)
0.11	(más de 500, 11%)

Ejemplo de salida:

**Compra en efectivo de 712 unidades a 12.04 c/u.****Precio neto: 8572.48****Descuento por mayoreo (11%): 942.97.****Descuento por pago en efectivo (5%): 381.48.****Total a pagar: 7248.06.**

- 8** Dado un número real que representa una cantidad de dólares inferior a \$9,999.99, exhibase la cantidad con signo de dólares y una coma en caso de requerirse. No se incluyan espacios en blanco entre el signo de dólares y el primer dígito.

Ejemplo de entrada:

25.04      4012.04

Ejemplo de salida:

\$25.04      \$4,012.04

- 9 Prodúzcase una tabla de verdad para una expresión booleana arbitraria con tres variables. Es decir, dada la expresión booleana (pero no como dato de entrada), prodúzcase una tabla que muestre el valor de la expresión para todas las combinaciones posibles de las variables.

Ejemplos: Supóngase que la expresión booleana es

**a AND (NOT b OR c)**

La salida deberá entonces ser parecida a ésta:

a	b	c	expresión
F	F	F	F
F	F	V	F
F	V	F	F
F	V	V	F
V	F	F	V
V	F	V	V
V	V	F	F
V	V	V	V

### ★ ★ ★ PROBLEMAS ESTIMULANTES

- 10 Dado un conjunto de tres dígitos decimales en la escala de cero a ocho, determínese la cantidad de números decimales únicos que se pueden formar, con sustitución, suponiendo que los seises se pueden invertir para formar nueves. Un problema más difícil es exhibir también los enteros.

Ejemplo de entrada:

0      3      6

Ejemplo de salida:

**19 enteros únicos**

(Específicamente 0, 3, 6, 30, 36, 39, 60, 63, 90, 93, 306, 309, 360, 390, 603, 630, 903 y 930.)

- 11 Los buzos con equipo autónomo deben realizar pausas para descompresión si se sumergen por períodos que excedan ciertos límites. La siguiente tabla muestra las pausas para descompresión en inmersiones a 70, 80 y 90 pies y los tiempos de descompresión requeridos.

PAUSAS PARA DESCOMPRESIÓN  
(TIEMPOS EN MINUTOS)

profundidad	tiempo en el fondo (en min)	a 30 pies	a 20 pies	a 10 pies
70 ft	100		...	33
	110		2	41
	120		4	47
	130		6	52
80 ft	100		11	46
	110		13	53
	120		17	56
	130		19	63
90 ft	100		21	54
	110		24	61
	120		32	68
	130	5	36	74

Los datos de entrada contienen la profundidad (en pies) y la duración (en minutos) de la inmersión. Determinense los tiempos de descompresión apropiados y exhibáse. Supóngase que los datos de entrada corresponden a uno de los renglones de la tabla. Inclúyase el siguiente mensaje en la salida: "PELIGRO": NO SE SUMERJA SIN LOS CONOCIMIENTOS APROPIADOS"

Ejemplo de entrada

80        120

Ejemplo de salida:

Para una inmersión a 80 pies durante 120 minutos se requieren las siguientes pausas para descompresión:

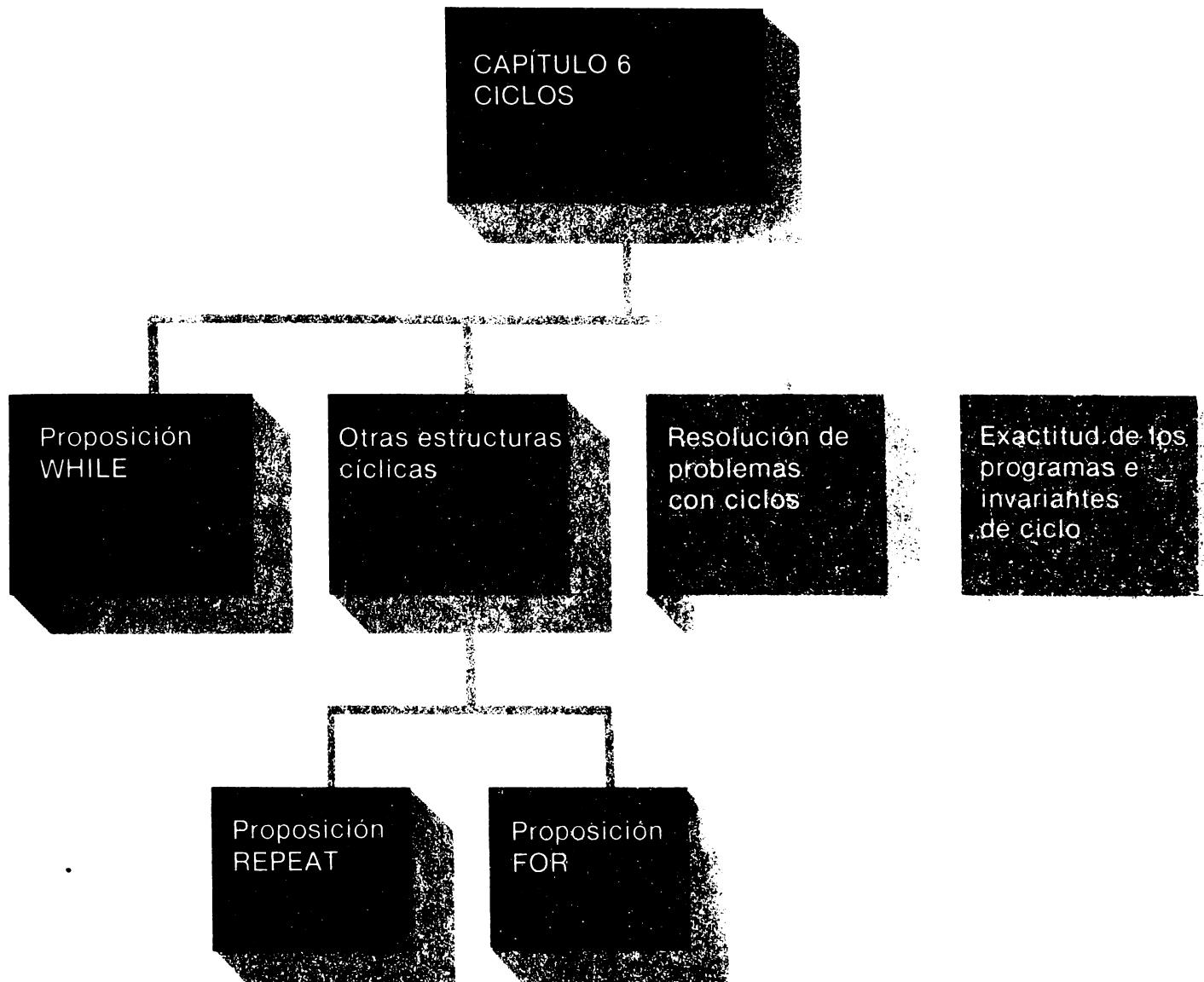
17 minutos a 20 pies

56 minutos a 10 pies

PELIGRO: NO SE SUMERJA SIN LOS CONOCIMIENTOS APROPIADOS



# CAPÍTULO 6



## CICLOS

**OBJETIVOS**

Después de completar este capítulo, el lector deberá ser capaz de:

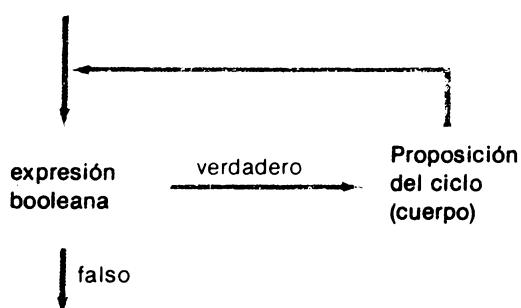
- Reconocer y aplicar la proposición WHILE a problemas que usen ciclos
- Reconocer y aplicar la proposición REPEAT-UNTIL a problemas que usen ciclos
- Reconocer y aplicar la proposición FOR a problemas que usen ciclos
- Aplicar la estrategia de resolución de problemas a problemas que usen ciclos
- Probar y depurar un programa en Pascal que use ciclos
- Opcionalmente, demostrar la exactitud de un ciclo

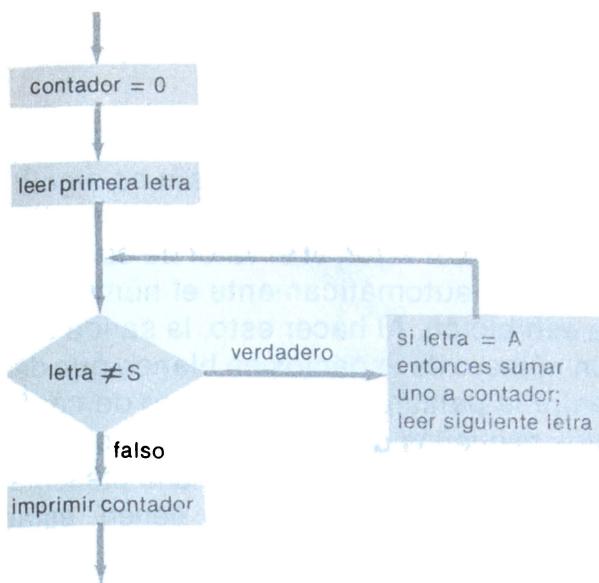
**PANORAMA GENERAL DEL CAPÍTULO**

El lector ya ha visto que la computadora es capaz de ejecutar repetidamente una secuencia de instrucciones en forma precisa. En este capítulo se hablará de la forma de controlar la ejecución repetida de una secuencia de proposiciones mediante una técnica de programación llamada **ciclo**. Se estudiarán en particular las estructuras de control de ciclos que permitan la ejecución repetida de una serie de proposiciones mientras se cumpla una condición. Estas condiciones se seguirán representando mediante expresiones booleanas. En la figura 6-1 se muestra una forma de una estructura de control cíclica.

En este caso, si se cumple la expresión booleana, se ejecutará una serie de proposiciones (conocida como **cuerpo** del ciclo). Después de la ejecución se vuelve a probar la expresión booleana. Mientras ésta se cumpla, se ejecutará una y otra vez el cuerpo del ciclo; cuando se deje de cumplir, el control pasará a la proposición que sigue al ciclo. Es importante darse cuenta de que la expresión booleana se evaluará y probará después de cada ejecución del cuerpo del ciclo. La secuencia de eventos en este ciclo será entonces “probar, ejecutar, probar, ejecutar, . . . , probar”. Por ejemplo, considérese el siguiente problema:

**Figura 6-1** Estructura de control cíclica.





**Figura 6-2** Flujo de control para el problema ejemplo.

*Obiéngase de los datos de entrada una lista con un número desconocido de calificaciones de letra (A, B, C, D o F), la cual termina con la letra S. Determinese cuántas de esas calificaciones son A.*

El flujo de control para este problema se muestra en la figura 6-2. La variable *contador* encontrará el número de calificaciones A. Al principio, el valor de *contador* debe ajustarse a cero, ya que no se ha encontrado ninguna calificación de A. Después se lee la primera calificación. En seguida comienza el ciclo. “Mientras” (“While”, en inglés) la calificación que se acaba de leer no sea igual a S, hacer (“do”) lo siguiente: si la calificación es A, sumar uno a *contador*; después, leer la siguiente letra. El control pasa al comienzo del ciclo, y éste se ejecutará mientras la calificación no sea S. En el caso de que se lea una calificación de S, el cuerpo del ciclo se pasará por alto y el control pasará a la proposición de impresión que sigue al ciclo.

El Pascal cuenta con varias construcciones cíclicas. A continuación se muestra una solución a este problema que utiliza la proposición WHILE de Pascal (supóngase que se declaró *contador* como entero y *letra* como de carácter):

```

contador := 0;
read (letra);
WHILE letra <> 'S' DO
BEGIN
  IF letra = 'A'
  THEN contador := contador + 1;
  read (letra)
  
```

```
END;
writeln ('El número de calificaciones A es ', contador: 1)
```

Se empleó una anchura de campo de uno en la proposición `writeln` para exhibir el valor de `contador`. Recuérdese (del Cap. 3) que esto *no* significa que habrá menos de 10 calificaciones A, sino que se quiere usar el mínimo posible de columnas para exhibir el valor; si el valor de `contador` es mayor que nueve, Pascal aumentará automáticamente el número de columnas que se utilizan para la exhibición. Al hacer esto, la salida que produce el programa no tendrá un número de espacios en blanco que dependa de la versión de Pascal entre la palabra `es` y el número de calificaciones A. Es preciso cuidar estos pequeños detalles para producir soluciones que no sólo sean correctas, sino que también tengan una presentación apropiada para los usuarios. Los programas que tienen estas características se suelen llamar *amables con el usuario*.

En este capítulo se estudiarán las estructuras de control cíclicas. Se darán detalles, en particular, de las proposiciones de ciclos en Pascal `WHILE`, `REPEAT-UNTIL` y `FOR` con aplicaciones a la resolución de problemas. También se incluye una sección opcional que trata la exactitud de los programas y las invariantes de ciclos.

## SECCIÓN 6.1 LA PROPOSICIÓN WHILE

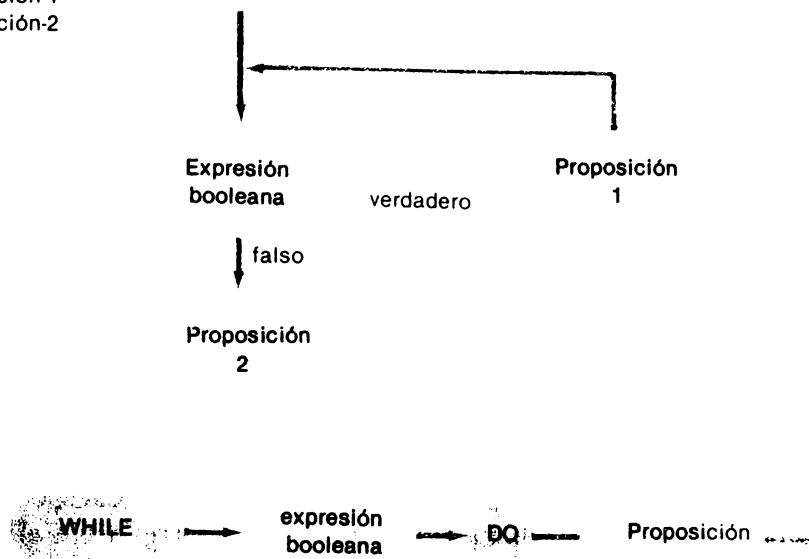
La proposición `WHILE` que se ilustró en el ejemplo anterior es la primera construcción cíclica que se analizará. La proposición `WHILE` en Pascal es un ejemplo de estructura de control de ciclos que tiene la siguiente forma general:

```
WHILE expresión-booleana DO
  proposición-1;
  proposición-2
```

La palabra reservada `WHILE` va seguida de una expresión booleana y después de la palabra reservada `DO`. Mientras se cumpla la expresión booleana, se ejecutará la proposición-1 (o un grupo de proposiciones delimitado por una pareja `BEGIN-END`). La proposición `WHILE` se ejecutará en forma repetida hasta que, al evaluarse la expresión booleana, resulte ser falsa, momento en el cual el control pasará a la proposición que sigue a la proposición `WHILE` (proposición-2). El diagrama de sintaxis y el flujo de control de la proposición `WHILE` se muestran en la figura 6-3.

Examíñese esta secuencia de proposiciones en Pascal (donde `núm` y `contador` se declaran como enteras):

```
Contador := 0;
read (núm);
WHILE núm > 0 DO
BEGIN
```

**Figura 6-3 Proposición WHILE.**

```

  contador := contador + 1;
  read (núm)
END;
writeln ('Número de enteros positivos = ', contador:1)
  
```

Cuando se ejecuta esta secuencia de proposiciones en Pascal se hace lo siguiente: se ajusta *contador* inicialmente a cero, se lee un entero (*núm*), se ejecuta el ciclo WHILE en forma repetida en tanto *núm* sea positivo. Este segmento de código en Pascal cuenta el número de enteros positivos que se capturan como datos de entrada. Cuando se capture un cero o un número negativo, se pasa por alto el cuerpo del ciclo y se ejecuta la proposición *writeln*. Obsérvese que el cuerpo del ciclo contiene dos proposiciones, por lo que éstas deben ir flanqueadas por una pareja BEGIN-END. Supóngase que los datos de entrada son: 12, 25, 10 y -9; con ellos el segmento de programa se ejecutará según se describe en el siguiente seudocódigo.

	<i>seudocódigo</i>	<i>explicación</i>
PASO 1	contador := 0	Inicializar en cero el contador
PASO 2	read (núm)	Leer el primer número
PASO 3	WHILE (núm > 0) DO	Probar si <i>núm</i> > 0. Si es así, continuar con el paso 4. Si no, seguir con el paso 7
PASO 4	sumar 1 a contador	Incrementar el contador
PASO 5	leer (núm)	Leer el siguiente número
PASO 6	volver al Paso 3	Para evaluar y probar la expresión booleana
PASO 7	exhibir (contador)	Exhibir el resultado

Obsérvese que los pasos tres a seis se ejecutarán en tanto el número capturado sea positivo. Al leerse el -9 (después de pasar tres veces por el ciclo), la evaluación de la expresión booleana “núm > 0” da lugar a un *false* y el control pasa a la proposición *writeln*. El valor del contador en ese momento será tres. La siguiente tabla muestra los valores que se asignarán a las variables *contador* y *núm* durante la ejecución de este segmento de programa.

<i>paso</i>	1	2	3	4	5	6	3	4	5	6	3	4	5	6	3	7
contador	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3	3
núm	12	12	12	25	25	25	25	10	10	10	10	-9	-9	-9	-9	-9

Como ya se vio, la prueba a mano de un algoritmo con valores prácticos puede ser de mucha ayuda para probar y depurar los programas.

Es importante comprender que la proposición *read* dentro del cuerpo del ciclo es esencial: sin ella el programa sería un *ciclo infinito* (sin fin), es decir, el programa nunca podría salir del ciclo. Así, sin la proposición *read*, la siguiente proposición WHILE sería un ciclo infinito (suponiendo que *núm > 0*):

```
WHILE núm > 0 DO
BEGIN
    contador := contador + 1
END
```

En este ciclo el valor de la expresión booleana jamás cambia. Por tanto el ciclo se ejecutará indefinidamente.

Supóngase que antes de escribir el programa ya se sabe que se van a leer 20 enteros, y se desea contar y exhibir el número de enteros positivos. En este caso, la proposición WHILE necesita verificar el número de veces que se ha ejecutado el cuerpo del ciclo. Puede utilizarse una variable llamada *variable de control de ciclo* para lograrlo. El siguiente segmento en Pascal realizaría ese trabajo (supóngase que *i*, *contador* y *núm* se declararon como enteros):

```
Contador := 0;          (* Inicializar el contador. *)
i := 1;                (* Inicializar la variable de control del ciclo. *)
WHILE i <= 20 DO      (* Continuar hasta que i > 20. *)
BEGIN
    read (núm);        (* capturar el siguiente dato de entrada. *)
    IF núm > 0
        THEN contador := contador + 1;
    i := i + 1         (* Incrementar la variable de control del ciclo. *)
END;
writeln ('El número de enteros positivos es', contador:1)
```

Es necesario asignar un valor inicial a la variable de control de ciclo *i* (en este caso uno) porque la computadora no es capaz de determinar el valor de *i* <= 20 sin conocer el valor de *i*. Es preciso cambiar el valor de la variable de control de

ciclo dentro del cuerpo del ciclo para evitar un ciclo infinito. En este caso, cuando  $i = 21$  el ciclo no se ejecutará más, ya que la expresión booleana “ $i \leq 20$ ” se vuelve falsa. No obstante, *contador* será menor o igual que 20. El control pasará en seguida a la proposición *writeln*.

### Ejemplo 6.1

*Examíñese las siguientes proposiciones en Pascal. ¿Qué se exhibirá y cuántas veces se ejecutará el cuerpo del ciclo?*

```
j := 0;
WHILE j < 5 DO
BEGIN
    write (j);
    j := j + 1
END
```

La salida es el valor de la variable de control de ciclo  $j$  al principio de cada ejecución del cuerpo del ciclo: 0, 1, 2, 3 y 4. El ciclo se ejecuta cinco veces.

¿Qué sucedería si se cambiara el orden de las dos instrucciones del cuerpo del ciclo? Es decir, ¿qué exhiben las siguientes proposiciones en Pascal?

```
''
j := 0;
WHILE j < 5 DO
BEGIN
    j := j + 1;
    write (j)
END
```

La salida sería entonces 1, 2, 3, 4 y 5. El cuerpo del ciclo se seguirá ejecutando cinco veces. Obsérvese que cuando  $j = 4$ , la expresión booleana se cumple y el cuerpo del ciclo se ejecuta. Después  $j = 5$  y se ejecuta la proposición *write*. Entonces se evalúa una vez más la expresión booleana. Esta vez, el valor de la expresión es *false* y la proposición WHILE habrá terminado su ejecución.

### Centinelas y ciclos

Un *centinela* es un valor especial que se emplea para indicar el final de una lista de datos. Por ejemplo, supóngase que se tiene una lista de calificaciones de estudiantes (cada una en la escala de cero a 100). El número de calificaciones no se conoce, pero la lista termina con el número —999 (el centinela). La figura 6-4 ilustra una lista de este tipo. Se puede usar una proposición WHILE para controlar el cuerpo del ciclo que obtiene las calificaciones de los datos de entrada. La expresión booleana de la proposición WHILE se utilizará para probar si ya terminó la lista mediante la detección del centinela.

## CALIFICACIONES

85
66
100
40
90
75
65
• •
• •
• •
65
-999      ← centinela

**Figura 6-4** Lista de calificaciones con centinela.

### Problemas 6.1

*En los datos de entrada aparecen una o más calificaciones de estudiantes en la escala de cero a 100, seguidas del valor centinela -999. Determinese y exhibase el total de las calificaciones excluyendo, naturalmente, al centinela.*

La solución a este problema aparece en seguida. Las variables *calif* y *total* supuestamente se declararon como enteras.

```
total := 0;
read (Calif);
WHILE calif <> -999 DO
BEGIN
    total := total + calif;
    read (calif)
END;
write ('El total de las calificaciones es', total:1)
```

Obsérvese que se asigna a *total* el valor inicial cero antes de la proposición WHILE. Dentro del cuerpo del ciclo, la proposición

*total := total + calif*

calcula los totales parciales de las calificaciones.

Cuando termina el ciclo, el contenido de la variable *total* será la suma de todas las calificaciones. Como ejercicio, se sugiere al estudiante verificar la ejecución de este segmento en Pascal para los datos de entrada

85            100            90            50            -999

*Este problema es el mismo que el problema 6.1 excepto que también se debe exhibir el promedio de las calificaciones.*

Para calcular el promedio de las calificaciones es preciso dividir el total de las calificaciones entre el número de calificaciones. Así, se deberá actualizar un contador dentro del cuerpo del ciclo que indique constantemente el número de calificaciones que se capturan. A este contador se debe asignar el valor inicial cero antes del ciclo WHILE. Supóngase que se declaró entera la variable *contador* y real una variable llamada *promedio*. El siguiente segmento de código en Pascal calcula y exhibe tanto el total de las calificaciones como su promedio.

```
total := 0;          (* dar valor inicial al total de calificaciones *)
contador := 0;        (* contar el número de calificaciones *)
read (calif);        (* capturar la primera calificación *)
WHILE calif <> -999 DO      (* probar el centinela *)
  BEGIN
    contador := contador + 1;      (* incrementar el contador *)
    total := total + calif;        (* sumar la nueva calificación *)
    read (calif)                  (* capturar la siguiente calificación *)
  END;
  write ('El total de las calificaciones es', total:1)
  (* Calcular la calificación promedio *)
  promedio := total / contador;
  writeln ('La calificación promedio es', promedio : 6 :2)
```

### Control de ciclos mediante variables booleanas

Es posible incluir expresiones booleanas compuestas en las proposiciones WHILE del Pascal mediante los operadores booleanos AND, OR y NOT. Las variables booleanas (en ocasiones llamadas *indicadores*) se pueden usar para controlar la ejecución de una proposición WHILE. Considérese el siguiente problema, en el que se prueba mas de una condición para controlar un ciclo.

### Problema 6.3

*Los datos de entrada contienen un solo entero positivo. Determiníse si este entero tiene divisores exactos y exhibase una indicación apropiada de lo averiguado. Un entero positivo tiene un divisor exacto si es divisible entre algún entero positivo diferente de él mismo y de la unidad. Por ejemplo, seis tiene los divisores exactos dos y tres. Siete, en cambio, no tiene divisores exactos.*

Se utilizará la variable *núm* para almacenar el entero positivo que se captura de los datos de entrada. Para resolver este problema se utilizará una estrategia de

“fuerza bruta” y se dividirá *núm* entre todos los enteros que se encuentran entre el uno y *núm*, excluyendo éstos. Es decir, se dividirá *núm* entre los enteros 2, 3, 4, . . . *núm* – 1. Si cualquiera de estos números divide en forma exacta a *núm*, *núm* tendrá un divisor exacto y se querrá terminar el ciclo que se emplee para probar todos los posibles divisores enteros.

Para hacer lo anterior se usará una variable llamada *indic* que se declarará como booleana. Se asignará el valor inicial *false* a esta variable, la cual indicará si *núm* tiene un divisor exacto o no. Si durante la ejecución del ciclo se determina que *núm* sí tiene un divisor exacto, se asignará a *indic* el valor *true*. Si *indic* vale *true* cuando se evalúe la condición booleana que controla el ciclo, se sabrá que ya se encontró un divisor exacto, por lo que no será necesario continuar el ciclo.

¿Qué sucede si se prueban todos los enteros entre uno y *núm* y no se encuentra un divisor exacto? La variable *indic* seguirá teniendo el valor *false*, pero será preciso terminar el ciclo. Es necesario probar también el cumplimiento de esta condición. Por tanto, la expresión booleana que se usará en la proposición WHILE tendrá dos partes: “NOT *indic*” y “*divisor < núm*”. El siguiente segmento en Pascal ilustra la solución.

```

read (núm);
indic := false; (* se supone que todavía no hay divisor exacto *)
divisor := 2;    (* se inicia la prueba de división con 2 *)
WHILE NOT indic AND (divisor < núm) DO
BEGIN
  indic := núm MOD divisor = 0;
  divisor := divisor + 1
END;
write ('El número', núm:1, ' ');
IF indic
THEN writeln ('tiene divisor exacto.')
ELSE writeln ('no tiene divisor exacto.')

```

#### La proposición de asignación

*indic := núm MOD divisor = 0*

puede parecer un poco extraña a primera vista. no obstante, recuérdese que la expresión

*núm MOD divisor = 0*

es una expresión booleana y producirá el valor *true* o *false* al evaluarse; el valor resultante se asignará entonces a la variable booleana *indic*. Otra forma de realizar esta asignación es mediante una proposición IF:

```

IF núm MOD divisor = 0
THEN indic := true
ELSE indic := false

```

En el momento en que se encuentre un divisor exacto (se asigne el valor *true* a *indic*) se terminará el ciclo, ya que la expresión booleana será falsa (“NOT *indic*” será falso). Además, si *núm* no tiene divisor exacto, el ciclo terminará cuando el divisor (que se incrementa en uno cada vez que se ejecuta el cuerpo del ciclo) tenga el valor de *núm*. La proposición IF-THEN-ELSE exhibirá la conclusión apropiada con base en el valor de *indic*. La razón principal de utilizar la variable booleana *indic* es terminar la ejecución del ciclo WHILE cuando se encuentre un divisor exacto. Sin esta variable booleana, el ciclo se ejecutaría una y otra vez hasta que el divisor fuera igual a *núm*. Esto es muy poco eficiente, especialmente si *núm* tiene un divisor exacto pequeño. Por ejemplo, si *núm* es un número par grande como 1200, entonces, con la variable *indic*, el cuerpo del ciclo WHILE se ejecutará una sola vez, ya que “1200 MOD 2 = 0”. En cambio, si no se cuenta con la variable *indic* para terminar el ciclo WHILE, ¡el cuerpo se ejecutará 1198 veces!

He aquí algunos puntos importantes que se deben tener en cuenta al usar la proposición WHILE:

- Se debe asignar un valor inicial a todas las variables de control del ciclo antes de comenzar éste.
- La expresión booleana de la proposición WHILE se evalúa antes de entrar al ciclo.
- La variable de control del ciclo se debe modificar dentro del ciclo para evitar un ciclo infinito.

### Proposiciones WHILE anidadas

Recuérdese que las proposiciones dentro del cuerpo de un ciclo WHILE pueden ser cualquier proposición o grupo de proposiciones en Pascal. De manera específica, se puede incluir otra proposición WHILE. Las proposiciones WHILE anidadadas son proposiciones WHILE dentro de otra proposición WHILE.

Se examinará ahora un ejemplo de lo anterior. Supóngase que se proporciona la población de cada una de las 25 ciudades más grandes de 10 diferentes estados y se desea identificar y exhibir la población de la ciudad más grande de cada estado.

Este problema se puede dividir fácilmente en dos subproblemas como sigue:

SUBPROBLEMA 1: Repetir el subproblema 2 para cada uno de los 10 estados.

SUBPROBLEMA 2: Encontrar y exhibir la población más grande de un estado.

Si se puede obtener la solución del subproblema 2, es posible resolver todo el problema. Puesto que se tienen exactamente 25 datos de población para cada estado, la solución del subproblema 2 incluye la lectura de las 25 poblaciones y la identificación y exhibición de la más grande. El código en Pascal que hace lo anterior se puede escribir en forma muy sencilla mediante un ciclo WHILE. Se supone que *población*, *mayor* e *i* se declaran todas como variables enteras.

```

mayor := 0;          (* la población más alta "por omisión" *)
i := 1;             (* dar valor inicial al contador de ciudades *)
WHILE i <= 25 DO    (* ¿faltan poblaciones por probar? *)
BEGIN
  read (población); (* capturar población de la siguiente ciudad *)
  IF población > mayor      (* ¿nueva población más alta? *)
  THEN mayor := población; (* sí, de manera que se almacena *)
  i := i + 1           (* incrementar contador de ciudades *)
END;                 (* del ciclo para un estado *)
writeln ('La población más alta es', mayor:1)

```

Puede observarse que a *mayor* se le asigna el valor inicial cero, y más adelante, cada vez que se capture una población más alta, se asigna este valor a *mayor*. El subproblema 1 requiere la repetición del subproblema 2 exactamente 10 veces. Esto se puede lograr si se incluye la solución del subproblema 2 en el cuerpo de un ciclo WHILE diseñado de manera que se ejecute 10 veces. Para crear un ciclo así es preciso emplear otra variable de control de ciclo entera, como *j*. Obsérvese que no es posible usar *i* para controlar este ciclo porque el código del subproblema 2 cambia su valor. La solución del problema completo tendrá este aspecto:

```

j := 1;              (* dar valor inicial al contador de estados *)
WHILE j <= 10 DO     (* ¿más estados? *)
BEGIN
  mayor := 0;          (* la población más alta "por omisión" *)
  i := 1;              (* contador de ciudades *)
  WHILE i <= 25 DO    (* ¿más poblaciones? *)
BEGIN
  read (población); (* capturar población de la siguiente ciudad *)
  IF población > mayor      (* ¿nueva población más alta? *)
  THEN mayor := población; (* sí, de manera que se le almacena *)
  i := i + 1           (* incrementar contador de ciudades *)
END;                 (* del ciclo de ciudades *)
writeln ('La población más alta es', mayor:1)
j := j + 1           (* incrementar el contador de estados *)
END                   (* del ciclo de estados *)

```

Obsérvese que, cada vez que se termina de ejecutar el ciclo WHILE interior, se vuelven a asignar los valores cero y uno a *mayor* e *i*, respectivamente.

## EJERCICIOS DE LA SECCIÓN 6.1

- 1 Supóngase que el siguiente código es parte de un programa correcto en Pascal.

```

x := 10;
WHILE x > 0 DO x := x - 3;
writeln (x)

```

Determine la salida exacta después de la ejecución de este código.

```

suma := 0;
i := 3;
WHILE i <= 7 DO
BEGIN
    suma := suma + i;
    i := i + 2
END;
writeln (i, suma)

```

- 3 Dados los datos de entrada

10      5      12      -5

y el segmento de código

```

suma := 0;
positivo := true;
WHILE positivo DO
BEGIN
    read (x);
    IF x < 0
        THEN positivo := false
    ELSE suma := suma + x
END;
writeln (suma)

```

¿qué se exhibirá cuando se ejecute el código?

- 4 Dados los datos de entrada

10      20      0      30

y el segmento de código

```

suma := 0;
i := 0;
read (x);
WHILE x > 0 DO
BEGIN
    i := i + 1;
    suma := suma + x;
    read (x)
END;
writeln (suma, i, x)

```

¿qué se exhibirá cuando se ejecute el código?

- 5 ¿Qué valores se exhiben al ejecutarse el siguiente segmento de programa?

```

context := 1;
WHILE context < = 3 DO
BEGIN
    contint := 5: ;
    WHILE contint < context DO;
    contint := contint -1;
    context := context + 1
END;
writeln (context, contint)

```

- 6 ¿Cuál será el contenido de *suma* y *valor* después de ejecutarse el siguiente segmento de programa? Supóngase que los datos de entrada son 5, 6, 7, -3, -4, 0, 5, 8 y 9.

```

n := 8;
suma := 0
i := 1;
indic := false;
WHILE (i < = n) AND NOT indic DO
- BEGIN
    read (valor);
    IF valor > 0
    THEN suma := suma + valor
    ELSE IF valor = 0
        THEN indic := true;
    i := i + 1
END;
writeln ('Fin de la prueba', suma, valor)

```

## SECCIÓN 6.2 OTRAS ESTRUCTURAS CÍCLICAS

Es posible que se presenten situaciones en las que se desea que un ciclo se ejecute por lo menos una vez *antes* de probar si se debe repetir o no la ejecución del ciclo. En la proposición WHILE, si el valor de la expresión booleana inicialmente es falso, el cuerpo del ciclo no se ejecutará. Por ejemplo, examíñese el siguiente segmento en Pascal (supóngase que *núm* y *contador* se declaran como variables enteras).

```

contador := 0;
read (núm);
WHILE núm > 0 DO
BEGIN
    contador := contador + 1;
    read (núm)
END

```

Si el primer número que se lee es negativo o cero, el cuerpo del ciclo WHILE no se ejecutará. La proposición WHILE prueba el valor de la expresión booleana primero para determinar si se debe ejecutar o no el cuerpo del ciclo. En algunos casos pudiera ser necesario ejecutar primero el cuerpo del ciclo y después llevar a cabo la prueba. Esta estructura de control cíclica está disponible en Pascal y es la proposición REPEAT-UNTIL. En estos ciclos se ejecuta primero el cuerpo del ciclo y en seguida se evalúa la expresión booleana.

### **Proposición REPEAT-UNTIL**

La proposición REPEAT-UNTIL tiene la siguiente forma general:

```

REPEAT
  proposición;
  proposición;
  .
  .
  .
  proposición
UNTIL expresión booleana

```

La palabra reservada REPEAT va seguida de una proposición en Pascal que forma el cuerpo del ciclo. Esta proposición se ejecuta en forma repetida hasta que el valor de la expresión booleana sea *true* (verdadero). Así, el ciclo REPEAT-UNTIL se ejecuta en tanto el valor de la expresión booleana sea *false*, exactamente al contrario que la proposición WHILE. Como siempre, es posible colocar un grupo de proporciones en el cuerpo del ciclo, pero *no* se necesita una pareja BEGIN-END después de la palabra REPEAT si se especifica más de una proposición. Las palabras REPEAT y UNTIL realizan de manera efectiva la función de *delimitación*. La figura 6.5 muestra el diagrama de sintaxis y el flujo de control de la proposición REPEAT-UNTIL.

Examíñese el siguiente segmento en Pascal (supóngase que *núm* y *contador* se declararon como variables enteras):

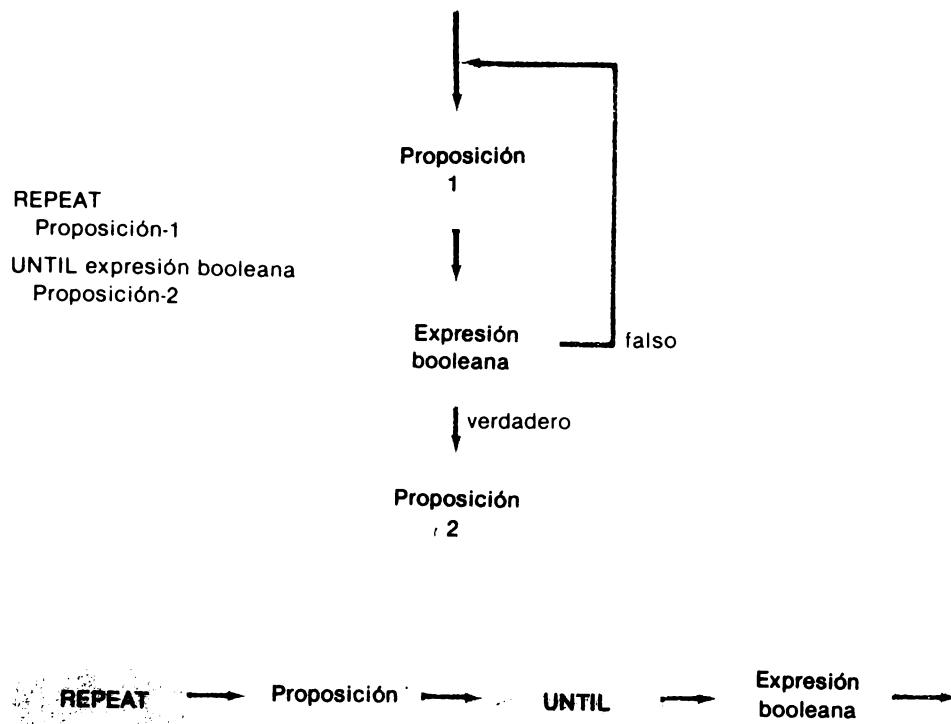
```

contador := 1;
REPEAT
  read (núm);
  contador := contador + 1
UNTIL contador > 20;
writeln ('Se leyeron veinte números.')

```

En esta secuencia de proposiciones, el cuerpo del ciclo REPEAT se ejecuta hasta que *contador* pasa de 20. Esto sucederá después de 20 ejecuciones del cuerpo del ciclo, y luego se ejecuta la proposición *writeln*. Obsérvese que no se requiere una pareja BEGIN-END.

En el siguiente ejemplo se debe buscar el primer dígito en una secuencia de caracteres que aparece en los datos de entrada.



**Figura 6-5** Proposición REPEAT-UNTIL.

### Ejemplo 6.2

*Los datos de entrada contienen una secuencia de caracteres. Determinese el primer dígito decimal incluido en estos datos y exhibase con un mensaje apropiado.*

En este problema se debe leer por lo menos un carácter antes de que se pueda realizar prueba alguna y, si la prueba resulta negativa (es decir, el carácter que se lee no es un dígito decimal), se deberá leer de nuevo. Este requisito de realizar alguna acción en forma repetida hasta cumplir una condición se pone en práctica de manera ideal en Pascal mediante la proposición REPEAT-UNTIL. Supóngase que se declaró una variable de carácter llamada *car*. En tal caso la solución aprovechará el ordenamiento de los caracteres:

```

REPEAT
  read (car)
  UNTIL (car > = '0') AND (car < = '9');
  writeln ('El primer dígito decimal es', car)
    
```

Recuérdese que el Pascal estándar exige que los dígitos decimales estén ordenados de tal manera que '0' sea menor que '1', '1' sea menor que '2' y así sucesivamente. Además exige que no se encuentren otros caracteres entre los dígitos decimales dados. Así pues, si *car* es mayor o igual que '0' y menor o igual que '9', se habrá verificado que se trata de un dígito decimal.

En seguida se atacará el mismo problema, pero se intentará utilizar un ciclo WHILE en vez de un ciclo REPEAT-UNTIL. Como se indicó anteriormente, es preciso leer un carácter antes de poder probar si se trata de un dígito decimal, por lo que la solución será parecida a ésta:

```
read (car);
WHILE (car < '0') OR (car > '9') DO
  read (car)
writeln ('El primer dígito decimal es', car)
```

Si se comparan las dos estructuras de control cíclicas anteriores, puede verse que el opuesto de la expresión booleana “(car < ‘0’) OR (car > ‘9’)” es la expresión “(car > = ‘0’) AND (car < = ‘9’)”. En general, si  $P$  y  $Q$  son expresiones booleanas, entonces “NOT (P AND Q)” es lógicamente idéntica a “NOT P OR NOT Q”. Además, “NOT (P OR Q)” es lógicamente idéntica a “NOT P AND NOT Q”. Estas dos propiedades se conocen como la *ley de DeMorgan*.

Como ejemplo adicional, el siguiente segmento en Pascal emplea una proposición REPEAT-UNTIL para exhibir los números del uno al 100:

```
núm := 1;
REPEAT
  writeln (núm);
  núm := núm + 1
UNTIL núm = 101
```

El ejemplo final muestra la forma como se puede usar un ciclo REPEAT-UNTIL en la validación de datos. En este caso el programa espera un número de mes en la escala de 1 a 12. Si el valor que se obtiene del dispositivo de entrada es incorrecto, se exhibe un mensaje apropiado para el usuario y se captura un valor adicional.

```
write ('Por favor escriba el número del mes:');
REPEAT
  readln (mes);
  IF (mes < 1) OR (mes > 12)
    THEN write ('Favor de dar un valor entre 1 y 12:')
  UNTIL (mes > = 1) AND (mes < = 12)
```

Esta técnica cíclica de verificación es muy útil cuando se emplea un sistema de cómputo interactivo, ya que permite la corrección inmediata de un error de entrada.

## La proposición FOR

Algunas veces se sabe por adelantado el número exacto de veces que se desea ejecutar las proposiciones de un ciclo. En este caso se puede usar la proposición FOR. La proposición FOR en Pascal es una estructura de control cíclica que eje-

cuta el cuerpo de un ciclo un número específico de veces y lleva automáticamente la cuenta del número de veces que se “pasa” por el cuerpo del ciclo. Por ejemplo, la siguiente proposición FOR hará que se lean  $n$  enteros y se calcule su suma (supóngase que  $n$ ,  $i$ ,  $núm$  y  $total$  se declararon como variables enteras):

```
total := 0;
FOR i := 1 TO n DO
BEGIN
    read (núm);
    total := total + núm
END
```

Si se emplea una proposición WHILE, es menester asignar un valor inicial a la variable de control del ciclo (en este caso  $i$ ) e incrementarla cada vez que se pasa por el cuerpo del ciclo. Así, se podría escribir el ejemplo anterior con una proposición WHILE de esta manera:

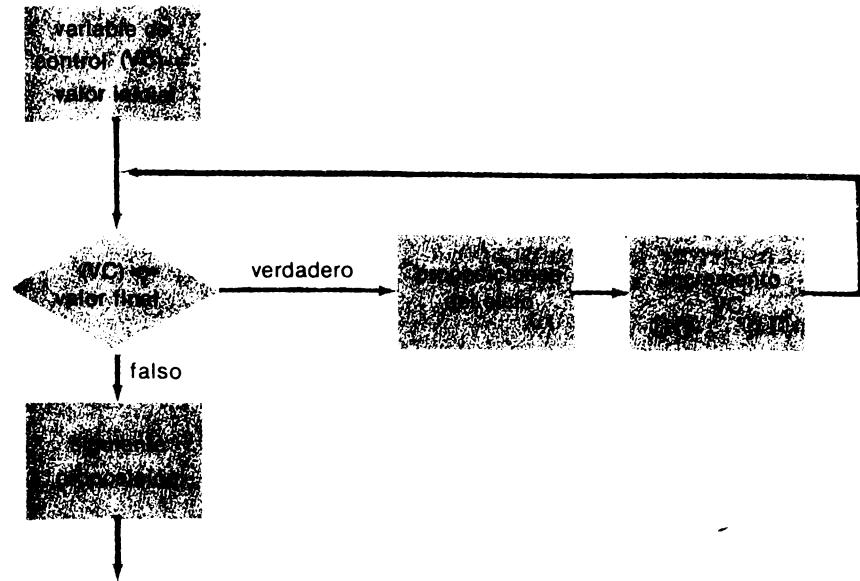
```
total := 0;
i := 1; (* dar valor inicial a la variable de control del ciclo *)
WHILE i <= 20 DO
BEGIN
    read (núm);
    total := total + núm;
    i := i + 1 (* incrementar la variable de control del ciclo *)
END
```

La proposición FOR en este ejemplo asigna el valor inicial 1 a la variable de control del ciclo y la incrementa cada vez que se llega al final del cuerpo del ciclo. Esto continúa hasta que la variable de control del ciclo llega a 20. En ese momento se ejecuta el cuerpo del ciclo una última vez y la ejecución sigue con la primera proposición después del ciclo.

Nótese que el valor de la variable de control del ciclo FOR (también llamada muchas veces *contador*) queda sin definir cuándo termina el ciclo. (No sucede así en el caso de los ciclos WHILE; el valor de la variable de control *sí* se conoce cuando el ciclo termina.) Cualquier variable de un tipo ordinal se puede emplear como variable de control de ciclo. La forma general de la proposición FOR es

**FOR contador := valor inicial TO valor final DO proposición**

La proposición debe comenzar con la palabra reservada FOR a la cual le sigue algo que parece una proposición de asignación que le da el valor inicial a la variable de conteo. Ésta va seguida de la palabra reservada TO y el valor final del contador. Por último aparece la palabra reservada DO, seguida de la proposición (o grupo de proposiciones delimitado por una pareja BEGIN-END) que constituye el cuerpo del ciclo. La variable de conteo, el valor inicial y el valor final deben ser todos del mismo tipo ordinal. Los pasos en la ejecución de una proposición FOR son (consúltense la Fig. 6-6):



**Figura 6-6 Estructura de control FOR.**

- PASO 1 Se asigna el valor inicial a la variable de conteo.
- PASO 2 Si la variable de conteo es menor o igual que el valor final, se ejecuta el cuerpo del ciclo, se sustituye el valor de la variable por su sucesor y se repite el paso 2. (En el caso de las variables enteras, el sucesor es simplemente el siguiente entero en secuencia. Los sucesores de otros tipos ordinales se estudian más a fondo en el Cap. 8.)
- PASO 3 Después de terminarse el ciclo FOR, el valor de la variable de conteo se hace “*indefinido*”. Es decir, no es posible suponer que la variable de conteo tiene un valor específico al término de la ejecución del ciclo.

### Ejemplo 6.3

*La proposición FOR*

**FOR i := 0 TO 5 DO write (i)**

*exhibe el valor de la variable de control durante cada ejecución del cuerpo del ciclo. Puesto que el cuerpo del ciclo se ejecuta seis veces, la salida incluirá los valores 1, 2, 3, 4 y 5.*

### Ejemplo 6.4

*La proposición FOR*

**FOR letra := 'A' TO 'E' DO write (letra)**

*también exhibirá el valor de la variable de control (letra, supuestamente declarada como variable de tipo char) durante cada ejecución del cuerpo del ciclo.*

*Esta vez el ciclo se ejecutará cinco veces y la salida incluirá los caracteres ABCDE. El sucesor de un valor de carácter es el carácter que le sigue inmediatamente en el ordenamiento lexicográfico (o secuencia de ordenamiento) del sistema de cómputo específico que se use.*

En un ciclo FOR también es posible decrementar la variable de conteo. Se emplea la palabra reservada DOWNT0 en vez de TO al escribir la proposición FOR. Por ejemplo, considérese la proposición

**FOR i := 10 DOWNT0 O DO write (i)**

Ésta exhibirá los valores 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 y 0, en ese orden. Los pasos para ejecutar un ciclo FOR con DOWNT0 son similares a los ya presentados:

- PASO 1 Se asigna el valor inicial a la variable de conteo.
- PASO 2 Si la variable de conteo es mayor o igual que el valor final, se ejecuta el cuerpo del ciclo, se sustituye el valor de la variable por su predecesor y se repite el paso 2.
- PASO 3 Después de determinarse el ciclo FOR, el valor de la variable de conteo se hace “indefinido”.

### Ejemplo 6.5

*Las variables de control con valor de carácter también se pueden decrementar. Examíñese el siguiente ciclo FOR:*

**FOR letra := 'e' DOWNT0 'a' DO write (letra)**

*La salida estará formada por las letras edcba.*

Para que se ejecute el cuerpo del ciclo FOR-TO-DO es preciso que el valor inicial sea menor o igual que el valor final. Del mismo modo, para que se ejecute el cuerpo de un ciclo FOR-DOWNT0-DO es necesario que el valor inicial sea mayor o igual que el valor final. No es un error especificar un ciclo FOR cuyo cuerpo jamás se ejecuta; en muchos casos el valor inicial o el final queda especificado mediante una expresión cuyo valor no se conoce antes de ejecutarse el programa.

### Ejemplo 6.6

*Considérese el siguiente ciclo FOR:*

**read (núm);  
FOR i := 10 TO núm DO write (i)**

Si *núm* es menor que 10, el cuerpo del ciclo FOR no se ejecutará.

Es importante subrayar que cualquier proposición FOR se puede sustituir por una serie equivalente de proposiciones de asignación y una proposición WHILE.

Pero *no* toda proposición WHILE puede sustituirse por una proposición FOR equivalente, ya que las proposiciones FOR deben especificar los valores iniciales y finales de la variable de control del ciclo. Es bueno recordar los siguientes puntos importantes acerca de las proposiciones FOR y los cuerpos de sus ciclos:

- La variable de control de un ciclo FOR puede ser de cualquier tipo ordinal (es decir, no real) y los valores iniciales y finales deben de ser mismo tipo.
- La variable de control del ciclo se debe declarar como variable local dentro del procedimiento que contenga al ciclo:

<i>correcto</i>	<i>incorrecto</i>
<pre>PROGRAM a (input,output); VAR i : integer; PROCEDURE b; VAR j : integer; BEGIN   FOR j := ...</pre>	<pre>PROGRAM x (input,output); VAR i : integer; PROCEDURE y; BEGIN   FOR i := ...</pre>

- La variable de control del ciclo no se puede cambiar dentro del ciclo. Además, no puede ser un parámetro variable formal para un procedimiento que contiene al ciclo FOR ni pasarse como parámetro variable verdadero a cualquier procedimiento que se invoque dentro del cuerpo del ciclo FOR. Por tanto, el siguiente segmento de código es incorrecto (supóngase que *proc* es un procedimiento que tiene un parámetro variable):

```
PROCEDURE mal (VAR i : integer);
BEGIN
  For i : 1 TO 10 DO proc (i)
END
```

- Los valores iniciales y finales de la variable de control de un ciclo FOR se pueden modificar dentro del cuerpo de un ciclo FOR sin cambiar el número de veces que se ejecuta el ciclo. Esto se debe a que los valores inicial y final se determinan y se guardan en variables "secretas" antes de la primera ejecución del cuerpo del ciclo. Por ejemplo, el siguiente ciclo exhibe los enteros del 1 al 10 aunque se modifica el valor final dentro del cuerpo del ciclo (supóngase que todas las variables son enteras):

```
final := 10;
FOR i:= 1 TO final DO
```

## BEGIN

```
final := 999;
write (i)
END
```

- Después de la ejecución de una proposición FOR, el valor de la variable de control queda totalmente indefinido. Es preciso asignar otro valor a la variable de control antes de intentar usarla en una expresión.
- Si el valor inicial y el final son iguales, el ciclo se ejecutará exactamente una vez. Por ejemplo, la siguiente proposición FOR exhibirá únicamente el número 1.

**FOR i := 1 to 1 DO write (i)**

- Los valores iniciales y finales para un ciclo FOR son expresiones arbitrarias que pueden incluir variables, constantes y operadores. Por ejemplo, el cuerpo del siguiente ciclo FOR se ejecutará exactamente seis veces (supóngase que todas las variables son enteras):

```
bajo := 0;
alto := 3;
FOR índice := bajo + 1 TO 2 * alto DO
BEGIN
    read (núm);
    write (núm)
END
```

- En este caso el valor de la variable *índice* irá de uno a seis. Los ciclos FOR se pueden anidar, al igual que las proposiciones WHILE y REPEAT-UNTIL. Así, las siguientes proposiciones FOR (tras declarar todas las variables como enteras) producirá seis renglones de salida que contiene los valores 1 1, 1 2, 1 3, 2 1, 2 2 y 2 3. Obsérvese que las variables de control de los ciclos FOR anidados no pueden ser las mismas.

**FOR i := 1 TO 2 DO
FOR j := 1 TO 3 DO writeln (i, j)**

Ahora se examinará un problema en el que puede utilizarse de manera efectiva la proposición FOR.

### Ejemplo 6.7

*Determinese y exhibase el valor de 1 + 2 + 3 + . . . + num, donde num es un entero mayor de cero.*

```
suma := 0;  
FOR i := 1 TO num DO suma := suma + i;  
writeln ('La suma es', suma:1)
```

Obsérvese que no se requiere un pareja BEGIN-END en el ciclo FOR, puesto que el cuerpo del ciclo sólo contiene una proposición.

En el siguiente problema se mostrará la forma como se puede usar la proposición FOR para simplificar el problema sobre el calendario del capítulo 5.

#### Problema 6.4

*Los datos de entrada contienen el número de mes y día del mes de un año no bisiesto. Calcúlese y exhibase el día del año correspondiente (en la escala de 1 a 365) mediante un ciclo.*

Recuérdese que este problema se resolvió en la sección 5.4 mediante una proposición CASE para seleccionar la suma de los días de los meses anteriores al mes en cuestión. La solución se puede simplificar con un ciclo FOR que calcule esta suma. Se supone que *mes*, *día*, *índice* y *díanual* son todos enteros, el problema se resolverá con este código en Pascal:

```
read (mes, dia); (* capturar número de mes y día *)  
díanual := 0; (* asignar valor inicial a la suma *)  
FOR índice := 1 TO mes - 1 DO  
  CASE índice OF  
    1, 3, 5, 7, 8, 10: díanual := díanual + 31;  
    2:                 díanual := díanual + 28;  
    4, 6, 9, 11:       díanual = díanual + 30  
  END;  
díanual := díanual + dia
```

Obsérvese que el ciclo FOR y el cuerpo de la proposición CASE sirven para determinar el número de días que hay en todos los meses anteriores al mes en cuestión (*mes*). Ésta es la razón de que se use “*mes - 1*” como valor final para el ciclo FOR. Es obvio que *índice* asumirá cada vez valores entre uno y “*mes - 1*”. Si se especifica el valor uno para *mes*, el cuerpo del ciclo FOR no se ejecutará. Puesto que *mes* no puede ser mayor que 12, la proposición CASE debe tomar providencias para todos los valores entre uno y 11. La última proposición suma a *díanual* el número de día del mes en cuestión.

#### ¿Estructuras de control definidas o indefinidas?

En este capítulo se han presentado tres estructuras de control en Pascal: WHILE, REPEAT-UNTIL y FOR. La proposición FOR establece lo que se conoce co-

múnmente como una *estructura de control definida*, ya que los valores iniciales y finales especificados para la variable de control del ciclo determinan de manera exacta el número de ejecuciones del cuerpo del ciclo. Al diseñar la solución de un problema conviene tener presentes las limitaciones del ciclo FOR:

- Es preciso determinar el número de veces que se va a ejecutar el cuerpo del ciclo antes de iniciar su ejecución.
- No es posible terminar “antes de tiempo” la ejecución de un ciclo FOR.

Por ejemplo, si se desea calcular el ingreso promedio de un grupo de 50 personas, es apropiado utilizar una proposición FOR, ya que se debe ejecutar exactamente 50 veces el cuerpo del ciclo (que captura un ingreso de los datos de entrada y lo agrega a la suma parcial).

En cambio, si no se conoce el número de personas que constituyen el grupo, será preciso utilizar una *estructura de control indefinida*, como la que se puede establecer mediante las proposiciones WHILE y REPEAT-UNTIL. Las estructuras de control indefinidas se pueden usar cuando el ciclo tiene las siguientes características:

- No se sabe necesariamente antes de la ejecución del ciclo el número de veces que se debe ejecutar el cuerpo del ciclo.
- Se puede utilizar más de una condición para terminar la ejecución del ciclo.

Recuérdese que el ciclo REPEAT-UNTIL siempre ejecuta el cuerpo del ciclo por lo menos una vez.

Es pertinente indicar que la proposición FOR no se debe usar cuando quizá se emplee más de una condición para terminar el ciclo. Considérese el siguiente problema.

### Problema 6.5

*Se sabe que no más de un saldo de cuenta de cheques en una lista de exactamente 1000 saldos es negativo. Encuéntrese y exhibase, si existe, el saldo negativo y su posición en la lista.*

El siguiente ciclo FOR resuelve este problema, pero es obvio que resulta ineficiente, ya que se deben procesar los 1000 saldos sin importar la posición o presencia del saldo negativo; se declara *saldo* como real e *índice* como entero.

```
FOR índice := 1 TO 1000 DO
BEGIN
    read (saldo);
    IF saldo < 0
    THEN writeln ('El saldo', índice:1, 'es' saldo:6:2)
END
```

En problemas como éste sería conveniente terminar la ejecución del ciclo una vez cumplida la condición especificada (en este caso el descubrimiento del saldo negativo o la prueba de todos los 1000 saldos). Muchos lenguajes de programación de alto nivel, entre ellos Pascal, cuentan con una proposición llamada **proposición de transferencia incondicional** que permite la continuación de la ejecución en un punto razonablemente arbitrario del programa. En Pascal, la transferencia incondicional se realiza mediante la proposición GOTO. En el problema del saldo negativo se podría usar una proposición GOTO para hacer que termine el ciclo y continúe la ejecución en otro punto. (En el apéndice D se encontrarán más detalles acerca de la proposición GOTO.) Empero, esta estrategia *definitivamente no se recomienda*, y la mayor parte de los programadores consideran casi cualquier uso de la proposición GOTO una práctica pésima.

Cuando se emplea más de una condición para determinar cuándo debe finalizar un ciclo, se debe usar una proposición WHILE o REPEAT-UNTIL. La solución del problema de saldos negativos se puede escribir con una proposición REPEAT-UNTIL así:

```

índice := 0;          (* asignar valor inicial a la posición en la lista *)
REPEAT
    read (saldo);           (* capturar un saldo *)
    índice := índice + 1    (* actualizar el índice *)
UNTIL (saldo < 0) OR (índice = 1000);
IF saldo < 0
THEN writeln ('El saldo', índice:1, 'es', saldo:6: 2)

```

Obsérvese que cuando se lee un saldo negativo el ciclo terminará (se cumplirá “saldo < 0”) y el control pasará a la proposición *writeln*. Nótese además que se podría haber usado una proposición WHILE en vez de la proposición REPEAT-UNTIL.

Puesto que Pascal cuenta con dos estructuras de control indefinidas (WHILE y REPEAT-UNTIL), ¿cuál se debe utilizar para un problema dado? En general no importa mucho cuál de las dos se elija, ya que son muy similares. Sin embargo, considérese la situación que se sugiere en la siguiente ilustración.

Los datos de ventas diarias de cierta compañía se deben procesar diariamente para proporcionar información oportuna de inventario. Se debe escribir un ciclo que procese los datos de cada venta. ¿Se debe usar un ciclo WHILE o un ciclo REPEAT-UNTIL, o resulta igual cuál de los dos se escoja? Se presentará un problema si no se realizan ventas en un día dado y se emplea una proposición REPEAT-UNTIL para crear el ciclo, ya que se requiere por lo menos una ejecución del cuerpo del ciclo. Es obvio que se debe usar la proposición WHILE, ya que se puede utilizar su expresión booleana para verificar que existan datos antes de continuar.

## EJERCICIOS DE LA SECCIÓN 6.2

- Determíñese el número de veces que se ejecuta el cuerpo del ciclo REPEAT-UNTIL en el siguiente segmento.

```

a := 6;
b := 5;
REPEAT
    a := a + 1
UNTIL a > b

```

- 2 Examíñese el siguiente segmento de programa:

```

i := 0;
REPEAT
    writeln ('Primitivo.')
UNTIL i = 0

```

Determinéñese la salida exacta cuando se ejecuta este segmento.

- 3 ¿Qué valor se exhibe cuando se ejecuta el siguiente segmento de código? Supóngase que *a* y *b* son variables enteras.

```

FOR b := 1 TO 3 DO
BEGIN
    IF b <= 1 THEN a := b - 1;
    IF b <= 2
        THEN a := a - 1
    ELSE a := a + 1
END;
writeln (a)

```

- 4 Examíñense los siguientes tres segmentos de código, cada uno de los cuales está incluido en un programa correcto en Pascal.

- I. FOR i := 1 TO 3 DO
   
    FOR j := i + 1 TO 3 DO
   
        write (i, j)
  
- II. i := 1;
   
j := 1;
   
WHILE (i <= 3) AND (j <= 2) DO
   
BEGIN
   
    write (i, j + 1);
   
    i := i + 1;
   
    j := j + 1
   
END
  
- III. FOR i := 1 TO 2 DO
   
        write (i, i + 1)

- a) Los tres segmento producen la misma salida.
- b) Los segmentos I y II producen la misma salida.
- c) Los segmentos II y III producen la misma salida
- d) Los segmentos I y III producen la misma salida.
- e) Ninguno de los segmentos produce una salida idéntica a la de otro.

5 Supóngase que se incluyó el siguiente código en un programa correcto.

```
bajo := 1;
FOR k := bajo TO 3 DO
BEGIN
  bajo := bajo + 2;
  write (k, bajo)
END
```

Determinese la salida que resulta de la ejecución de este código.

6 Determinese la salida que resulta de la ejecución del siguiente código:

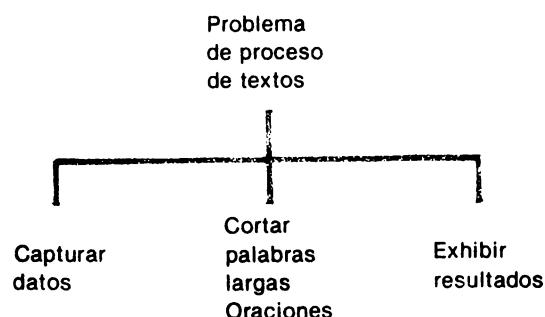
```
alto := 4;
FOR k := alto DOWNTO 3 DO
  writeln (k, alto)
```

### SECCIÓN 6.3 RESOLUCIÓN DE PROBLEMAS MEDIANTE CICLOS

En esta sección se aplicarán los ciclos a un problema específico. Estúdiense el siguiente problema de proceso de textos para resolución en computadora (véase la Fig. 6-7).

*Un estudiante terminó un trabajo final escrito para un determinado curso. Obténgase la siguiente información estadística acerca del trabajo, la cual ha pedido su profesor:*

**Figura 6-7** Diseño descendente del problema de proceso de texto



- Número de oraciones en el trabajo
- Número de palabras empleadas en el trabajo
- Número de promedio de palabras en cada oración
- Número de palabras de 10 letras o más.

Para simplificar el problema, supóngase que se utiliza un punto para marcar el final de cada oración. Así, la última palabra de cada oración irá seguida de un punto. Supóngase además que utiliza un signo de dólares como centinela para marcar el final del trabajo.

Este problema se puede dividir en tres subproblemas, como sigue:

SUBPROBLEMA 1: Leer los caracteres del texto.

SUBPROBLEMA 2: Contar el número de palabras, oraciones y palabras de 10 letras o más.

SUBPROBLEMA 3: Exhibir los resultados.

El problema de más difícil resolución es el primero, la lectura de los datos. Puesto que hay un número desconocido de caracteres terminados por un centinela ("\$"), se puede usar una proposición WHILE para buscar el final del trabajo. Se puede emplear otra proposición WHILE para buscar el final de una oración. Dentro de esta proposición WHILE se leerán y contarán las palabras empleadas en una oración dada.

El siguiente algoritmo escrito en pseudocódigo bosqueja la solución con mayor detalle:

*Algoritmo de proceso de texto*

Asignar valor inicial a los contadores.

Leer un carácter.

Mientras no sea fin de texto ("\$"):

    Mientras no sea fin de oración (".'"):

        Leer una palabra.

        Incrementar contador de palabras.

        Si longitud de palabra mayor de 10 letras, incrementar contador de palabras largas.

    Incrementar contador de oraciones.

    Leer un carácter.

Exhibir número de oraciones.

Exhibir número de palabras y palabras largas.

Calcular y exhibir promedio de palabras por oración.

El único paso que requiere un análisis más profundo es "leer una palabra". Puesto que las palabras están separadas por medio de caracteres especiales (como espacios en blanco, comas y signos de punto y coma) se puede hacer caso omiso de todos los caracteres hasta que se lea una letra mayúscula o minúscula. Así, la refinación de "leer una palabra" será:

Leer una palabra:

Desechar caracteres que no sean letras.

Leer letras.

Contar letras.

Ahora se puede escribir la primera versión del programa en Pascal llamado *estadistexto*.

```

PROGRAM estadistexto (input, output);
(* Determinar el número de palabras, oraciones, palabras largas *)
(* De 10 letras lo más) y promedio de palabras por oración en *)
(* los datos de entrada. El texto termina con el signo "$". *)
CONST centinela = "$";
VAR
  longitud : integer; (* longitud de la palabra actual *)
  palabras: integer; (* total de palabras leídas *)
  largas : integer; (* número de palabras largas *)
  oraciones : integer; (* número de oraciones *)
  promedio : real; (* palabras por oración *)
  car : char; (* carácter capturado *)
(*.....*)
BEGIN
  (* Asignar valor inicial a todos los contadores *)
  palabras := 0;
  largas := 0;
  oraciones := 0;
  (* Ciclo principal *)
  leer (car);
  WHILE car <> centinela DO (* no ha terminado el texto *)
    BEGIN
      WHILE car <> '.' DO (* no ha terminado la oración *)
        BEGIN
          (* pasar por alto caracteres que no sean letras *)
          WHILE NOT ((car >= 'A') (car <= 'Z'))
            AND NOT ((car >= 'a') AND (car <= 'z'))
            DO read (car);
          longitud := 0; (* comienza palabra nueva *)
          (* leer la palabra nueva *)
          WHILE (car = 'A') AND (car = 'Z') OR
            (car >= 'a') AND (car <= 'z') DO
            BEGIN
              longitud := longitud + 1;
              read (car)
            END;
          palabras := palabras + 1;
        END;
      END;
    END;
  END;

```

```

IF longitud > 10
THEN largas := largas + 1
END; (* del WHILE car <> '.' *)
oraciones := oraciones + 1;
read (car)
END; (* del ciclo principal *)
(* Exhibir resultados *)
writeln ('Número de oraciones : ', oraciones: 8);
writeln ('Número de palabras largas: ', largas: 8)
writeln ('Número de palabras largas: ', largas:8)
promedio := palabras / oraciones;
writeln ('Promedio de palabras por oración: ', promedio:8:2)
END.

```

---

*Programa estaditexto*

**Pienso, luego existo. Creo que fue Descartes quien dijo eso.  
Según la revista Artificial Intelligence, la computadoras pueden pensar.  
Luego, existen. \$**

Número de oración:	4
Número de palabras:	22
Número de palabras largas:	2
Promedio de palabras por oración:	5.50

*Programa estadistexto: ejemplo de ejecución*

El programa anterior se puede modificar para manejar otros problemas relacionados con la puntuación y la gramática (signos de interrogación, guiones, etc.) así como para revisar la ortografía de las palabras (véase el ejemplo del Cap. 4). También se puede emplear para verificar el cumplimiento de las especificaciones para el trabajo final de un curso, un artículo de revista y hasta un libro. En un capítulo posterior se estudiará con mayor detalle el manejo de datos de caracteres.

#### **SECCIÓN 6.4 EXACTITUD DE LOS PROGRAMAS E INVARIANTES DE CICLO (OPCIONAL)**

Después de escribir un programa para resolver un problema dado, ¿cómo se sabe con certeza absoluta que el programa resuelve el problema? Se pueden procesar muchos datos de entrada de prueba y verificar que la salida del programa es la correcta para esos casos. Pero aun así, es posible que el programa falle con otros datos de entrada que no se probaron. Si el programa es grande, sería muy poco práctico (si no imposible) verificar que sea correcto para todas las entradas posibles.

Los científicos de la computación en el área de la investigación teórica, conocida como *verificación de programas*, han inventado técnicas para establecer la

exactitud de los programas. En esta sección se examinarán algunas de estas técnicas, especialmente aquellas que se relacionan con la demostración de la exactitud de los ciclos. Las demostraciones de la exactitud pueden ser muchas veces complicadas y largas. No obstante, existen muchos casos en los que el costo de demostrar que un programa es correcto, es pequeño comparado con el costo que resulta de ejecutar un programa incorrecto. Piénsese en el peligro potencial de ejecutar un programa defectuoso que controle una planta de energía nuclear.

## Precondiciones y poscondiciones

El método de verificación de programas que se presenta aquí está basado en una serie de observaciones acerca de las variables de un programa antes, durante y después de la ejecución de un segmento de programa. Estas observaciones, llamadas *aserciones*, son afirmaciones acerca de las variables del programa que se espera sean ciertas. Estas aserciones del programa sirven para describir el estado del programa en cada paso de la ejecución. Además, las aserciones describen a las variables del programa y sus relaciones durante las distintas etapas de la ejecución del programa. Lo usual es expresar las aserciones en forma de expresiones booleanas. Los siguientes son ejemplos de aserciones encerradas entre llaves (delimitadores de comentarios) { y }.

```
{ núm es positivo }
{ x + y > z }
{ divisor <> 0 }
{ núm > 0 e índice < núm }
{ Contador es la suma de todos los enteros de 1 a 100 }
```

Para establecer la exactitud de un programa (o segmento de programa) se especifica en forma precisa la tarea que se va a realizar. Una forma de hacerlo es declarar la *precondición* y la *poscondición*. La precondición es una aserción que describe el estado de las variables del programa antes de comenzar la ejecución. Si no se sabe o se presupone algo acerca de las variables del programa, la precondición es la constante booleana *true*. La poscondición es una aserción que describe el estado de las variables del programa después de finalizar la ejecución. Si la precondición y la poscondición especifican correctamente la tarea que va a realizar un segmento de programa, ya se podrá considerar la cuestión de si el programa es correcto o no.

El siguiente ejemplo sencillo muestra un segmento de programa que es correcto (supóngase que *núm1* y *núm2* son variables enteras):

```
{ true }
num1 := 4;
num2 := 5
{ num1 = 4 y num2 = 5 }
```

Obsérvese que no se presupone cosa alguna antes de comenzar la ejecución del segmento de programa, por lo que la precondición es *true*. La poscondición

describe el estado de las variables *núm1* y *núm2* después de la ejecución del segmento de programa.

La estrategia divide y vencerás para resolución de problemas también se puede aplicar a la demostración de la exactitud de un programa. Se puede crear una prueba de exactitud de un programa si se le divide sucesivamente en una serie de pruebas de exactitud de segmentos más pequeños del programa. Si se puede demostrar la exactitud de estos segmentos de programa más pequeños, será posible combinarlos para obtener una prueba de exactitud del programa completo. Estúdiense el siguiente ejemplo, en el que *a*, *b*, *c* y *d* son variables enteras.

```
a := b DIV d;
c := a + b
```

La precondition de la primera proposición es la aserción que afirma que el divisor *d* no es cero:

$$\{ d \neq 0 \}$$

La poscondición de la primera proposición es la aserción

$$\{ a = b \text{ DIV } d \}$$

Esta poscondición actuará como precondition de la segunda proposición antes de su ejecución. Después de la ejecución de la segunda proposición, la poscondición de este segmento de programa es

$$\{ a = b \text{ DIV } d \text{ and } c = a + b \}$$

Esto equivale a la aserción

$$\{ c = b \text{ DIV } d + b \}$$

Así, si las dos partes del programa

$\{ d \neq 0 \}$	$\{ a = b \text{ DIV } d \}$
$a := b \text{ DIV } d$	$y$
$\{ a = b \text{ DIV } d \}$	$\{ c = b \text{ DIV } d + b \}$

son correctas, el programa

```
{ d <> 0 }
a := b DIV d;
c := a + b
{ c = b DIV d + b }
```

es correcto. Por lo supuesto anteriormente, es posible eliminar sin problema la aserción “ $\{ a = b \text{ DIV } d \}$ ” .

Los ejemplos que se presentaron hasta aquí fueron sencillos con el fin de ilustrar los conceptos de precondición y poscondición. La verificación de la exactitud de los ciclos es más difícil, dado que no basta con verificar que el ciclo hace lo que supuestamente debe hacer, sino que también se debe comprobar que terminará en un momento dado.

Las *invariantes de ciclo* son aserciones que se cumplen antes de entrar a un ciclo y que se siguen cumpliendo después de cada iteración del ciclo. Por tanto, las invariantes de ciclo no sufren cambios después de cada ejecución del cuerpo del ciclo. Las *variantes de ciclo* son aserciones que cambian cada vez que se pasa por el ciclo. Sirven para garantizar la eventual terminación del ciclo al alcanzar o pasar un valor llamado *umbral*. La invariante de ciclo relaciona a la precondición del ciclo con la poscondición. Como se verá, si se determinan en forma apropiada las aserciones invariante y variante, se podrán emplear para demostrar la exactitud del ciclo.

Considérese el siguiente segmento de programa que calcula *producto* (*número \* cuenta*) para dos enteros, *número* y *cuenta* (donde *número* no es negativo) por adición repetitiva. Es decir, se asigna el valor inicial cero a *producto* y en seguida se suma *cuenta* a *producto* en forma repetida. También se incluyen la precondición y la poscondición del segmento de programa.

```
{número > = 0}
índice := 0;
producto := 0;
WHILE índice < número DO
BEGIN
    producto := producto + cuenta;
    índice := índice + 1
END
{producto = número * cuenta}
```

Para demostrar que este ciclo es correcto con respecto a la precondición y a la poscondición, es preciso establecer una aserción invariante y una aserción variante correspondiente. Establecer la invariante de ciclo es muchas veces un paso difícil, dado que siempre hay más de una invariante para un ciclo dado. Por ejemplo, la aserción “{índice  $\geq 0$ }” se cumple antes del ciclo y después de cada ejecución del ciclo, de manera que es una invariante de ciclo. Pero esta invariante no es muy útil para demostrar la exactitud del ciclo. La invariante de ciclo deberá afirmar más información acerca de la acción del ciclo. Dado que ni *índice* ni *número* son negativos y están relacionados por medio de la condición del WHILE, “(índice < número)”, se concluye que la siguiente aserción es una invariante de ciclo útil:

{índice  $\leq$  número}

Obsérvese que para cualquier valor no negativo de *número* se cumplirá esta invariante de ciclo antes del ciclo y después de cada iteración de éste.

Si el ciclo es correcto, el valor de producto también deberá ser correcto. Por tanto, es preciso encontrar una invariante de ciclo relacionada con el cálculo de *producto* dentro del ciclo. La aserción

{*producto* = *índice* \* *cuenta*}

se cumplirá antes de entrar al ciclo y después de cada ejecución del ciclo, de suerte que es una invariante de ciclo útil. Ya es posible, pues, escribir la invariante final del ciclo:

{*índice* <= *número* y *producto* = *índice* \* *cuenta*}

que permite confiar en que el ciclo realizará correctamente la tarea de calcular *producto* mediante la adición repetida.

Ahora es preciso demostrar que el ciclo terminará en un momento dado, para lo cual se determinará la variante del ciclo. Puesto que “(*índice* <= *número*)” e *índice* aumenta cada vez que se pasa por el ciclo, el ciclo terminará cuando *índice* llegue al umbral, en este caso el valor de *número*. Por tanto, la aserción variante es

{índice < *número* e índice aumentan}

y garantiza que el ciclo tiene un final.

A continuación se muestra el segmento de programa con precondición y poscondición para cada proposición.

```
{número >= 0}
índice := 0;
{número >= 0 e índice = 0}
producto := 0;
{número >= 0, índice = 0 y producto = 0}
{índice <= número y producto = índice * cuenta}
WHILE índice < número DO
    {índice < número y producto = índice * cuenta}
BEGIN
    producto := producto + cuenta;
    {índice < número y producto = (índice + 1) * cuenta}
    índice := índice + 1
    {índice <= número y producto = índice * cuenta}
END
{producto = número * cuenta}
```

En este punto se puede concluir que el segmento de programa es correcto con respecto a la precondición “{número >= 0}” y la poscondición “{producto = número \* cuenta}”. La técnica para demostrar la exactitud de un programa que se describió en esta sección se puede resumir como sigue:

- Describir de manera precisa la precondition y poscondición del programa (o segmento de programa) con objeto de especificar la tarea que se va a realizar.
- Dividir el programa (o segmento de programa) en segmentos cada vez más pequeño, y describir la precondition y poscondición de cada segmento.
- Establecer la exactitud de cada uno de los segmentos de programa con respecto a su precondition y poscondición.
- Por último, concluir que el programa (o segmento de programa) es correcto con respecto a la precondition y postcondición originales.

La verificación de los programas requiere bastante trabajo, aun en el caso de programas elementales. Muchas veces es posible demostrar la exactitud de programas más complejos al recurrir a técnicas avanzadas de lógica y matemáticas. Por añadidura, se han creado programas de computadora que ayudan en el proceso de verificar la exactitud de un programa. No obstante, la verificación de programas es una disciplina que está todavía en pañales y, en la mayor parte de los casos, no se verifica formalmente la exactitud de los programas. Entonces, ¿Por qué molestarse en tratar de demostrar la exactitud de un programa? En algunos casos, como en la verificación de ciclos, esto puede ser beneficioso. Por ejemplo, permite documentar muy cuidadosamente el ciclo al incluir aserciones y asegura que el ciclo funcionará correctamente como se supone. Otra razón importante es que el conocimiento de las técnicas de verificación de programas puede ser muy útil cuando se aplican éstas a la resolución, prueba, depuración y documentación de los programas. Dicho de otra forma, puede contribuir a hacer de la persona un mejor programador. Por último, las pruebas nunca pueden demostrar la ausencia de errores, únicamente su presencia.

## SECCIÓN 6.5 ESTRUCTURAS DE CONTROL FUNDAMENTALES: RESUMEN

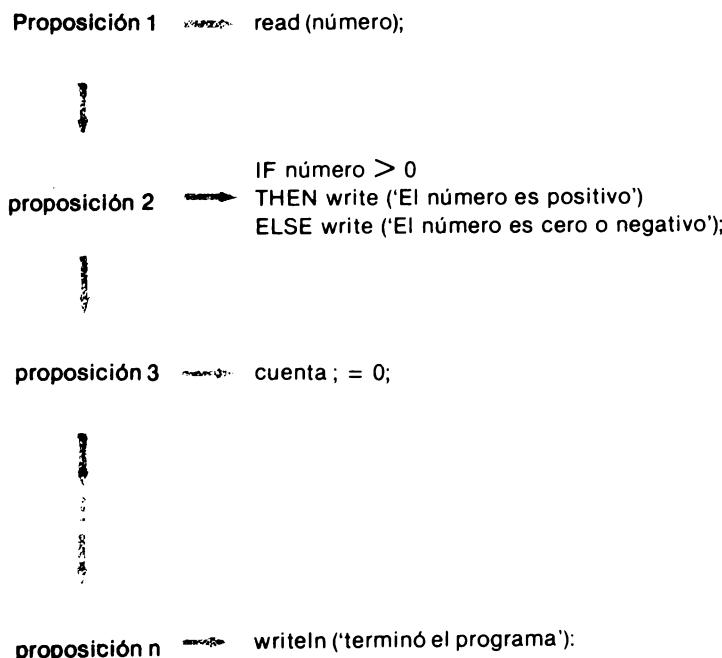
En este punto se recordarán los últimos seis capítulos y se hará un resumen de las estructuras de control fundamentales que se emplean en lenguajes de programación como Pascal. Hasta ahora se han estudiado las siguientes cuatro estructuras de control: 1) ejecución secuencial, 2) definición e invocación de procedimientos, 3) selección y 4) ciclos. Ya se cuenta con todas las estructuras de control necesarias para escribir programas de computadora estructurados. Gran parte del resto de este curso se ocupará de la aplicación de estas estructuras de control, las cuales están incluidas en Pascal, a la resolución de diversos problemas, especialmente aquellos que abarcan diferentes formas de representación y estructuración de los datos. Cabe mencionar que las estructuras de control son parte fundamental de casi todos los lenguajes de programación de alto nivel estructurados, pero pueden representarse en cada lenguaje mediante reglas de sintaxis distintas (véase la Fig. 6-8).

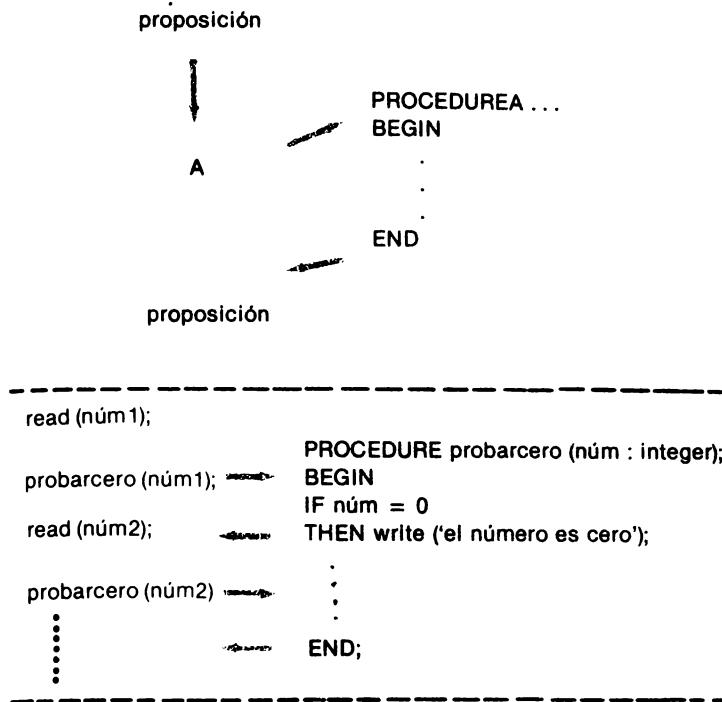
Al resumir las estructuras de control fundamentales se obtienen los conceptos siguientes:

- 1 La *ejecución secuencial* es la ejecución de una secuencia de proposiciones, una después de otra. En la figura 6-9, por ejemplo, se ejecutaría la proposición 1, después la proposición 2 y así sucesivamente, hasta ejecutarse la última.

**Figura 6-8** Estructuras de control fundamentales.

- 2 La declaración e invocación de *procedimientos* permite dar nombre a un grupo de proposiciones, y este nombre se puede usar posteriormente mediante la inclusión de una sola proposición (la invocación). Los parámetros verdaderos en el programa principal especifican los valores en el procedimiento que están asociados con los parámetros formales (*representantes*) y permiten emplear un procedimiento con diferentes valores cada vez que se invoca. La figura 6-10 ilustra un procedimiento con un ejemplo en Pascal.
- 3 La *selección* permite escoger diferentes cursos de acción de acuerdo con el valor de una expresión. La selección se realiza en Pascal mediante las proposiciones IF-THEN-ELSE, IF-THEN y CASE. La figura 6-11 ilustra la estructura de control IF-THEN.

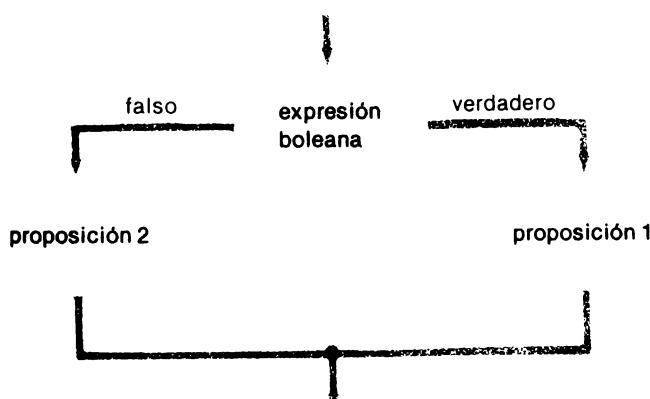
**Figura 6-9** Estructura de control secuencial y ejemplo en Pascal.



**Figura 6-10** Estructura de control de procedimiento y ejemplo en Pascal.

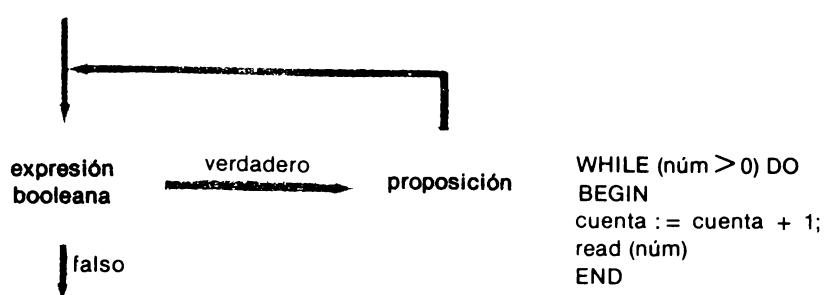
- 4 Las estructuras de control *cíclicas* permiten especificar la repetición controlada de una o más proposiciones. La repetición se puede controlar por medio de una expresión booleana (cuyo valor indica cuándo se debe continuar la repetición del cuerpo de un ciclo WHILE o REPEAT-UNTIL) o mediante una escala de valores ordinales (en una proposición FOR). Pascal cuenta con dos estructuras de control de ciclos indefinidas (los ciclos WHILE y REPEAT) y

**Figura 6-11** Estructura de control de selección y ejemplo en Pascal.



```

IF (núm >0)AND (núm < 11)
THEN write ('válido')
Else write ('no válido')
```



**Figura 6-12** Estructura de control cíclica y ejemplo en Pascal.

una estructura de control de ciclos definida (el ciclo FOR). La figura 6-12 ilustra la estructura de control WHILE del Pascal.

## SECCIÓN 6.6 TÉCNICAS DE PRUEBA Y DEPURACIÓN

### Ciclos infinitos

¿Cómo se puede determinar si un programa está en un ciclo infinito? Si el programa se está ejecutando y no termina, es posible que esté en un ciclo infinito. ¿Cuáles son las fuentes de error? La siguiente es una lista de los puntos que se podrían revisar:

- Asegurarse de que se asignó un valor inicial a la variable de control antes de comenzar cada ciclo WHILE.
- Asegurarse de que se modifica la variable de control dentro del cuerpo del ciclo.
- Verificar que no siempre se cumpla la expresión booleana. Por ejemplo, la proposición

**WHILE (1 + 1 = 2) DO**

siempre provocará un ciclo infinito.

- Verificar que no haya una proposición vacía inmediatamente después del DO. Por ejemplo, supóngase que *núm* es inicialmente mayor que 10. Entonces la proposición

**WHILE (núm > 10) DO;**

es un ciclo infinito, ya que la variable de control *núm* nunca cambia. Si se coloca por equivocación un signo de punto y coma inmediatamente después del DO, el compilador de Pascal no informará de un error de sintaxis. Se supondrá que la proposición vacía es la proposición que se desea ejecutar repetidamente.

- En las proposiciones REPEAT-UNTIL es preciso asegurarse de que la expresión booleana no es siempre falsa. Considérese el siguiente ejemplo:

```
REPEAT
    proposición
UNTIL (núm > 1) AND (núm < 0)
```

Aquí, “(núm > 1) AND (núm < 0)” nunca se cumple, por lo que la proposición REPEAT será un ciclo infinito.

## Errores de ciclos

Un error al codificar ciclos es el error de “falta uno”. Examíñese el siguiente ejemplo, que supuestamente debe exhibir los números nones del uno al 11.

```
núm := 1;
REPEAT
    writeln (núm);
    núm := núm + 2
UNTIL (núm = 11)
```

La salida que se produce en realidad consta de los números 1, 3, 5, 7, y 9. El número non 11 no se exhibe, puesto que la expresión booleana se cumple cuando  $núm = 11$ , y el ciclo termina. Se puede corregir el programa si se cambia la expresión booleana a  $núm = 13$ . ¿Qué sucedería si se cambia la condición a  $núm = 12$ ? Probar a mano unas cuantas iteraciones del ciclo demostrará al lector que está en un ciclo infinito.

Al escribir ciclos, es conveniente que el programador compruebe que se entra al ciclo en las condiciones apropiadas. Debe asegurarse de que se asigne un valor inicial a la variable de control antes del ciclo y de que el ciclo termine en algún momento. Para verificar la ejecución del ciclo basta probar unos cuantos valores a mano. No hay que olvidar la revisión de los valores iniciales y finales. Éstos se llaman *condiciones de frontera*. Muchos errores de los ciclos se deben a que el valor inicial es incorrecto o bien a que nunca se llega al valor final.

A continuación se da un ejemplo de un ciclo al que se entra de manera incorrecta. El segmento de programa debería leer un número desconocido de números no negativos, almacenar el total de esos valores y terminar cuando se lea un valor negativo, sin sumar el número negativo al total. Supóngase que *número* y *total* son variables enteras.

```
total := 0;
WHILE número > = 0 DO
BEGIN
    read (número);
    total := total + número
END
```

La forma de entrar a este ciclo es incorrecta porque el valor de la variable *número* no está definido. Supóngase que se da un valor inicial a la variable *número* para poder entrar al ciclo. En ese caso, ¿se resolverá el problema con el siguiente código?

```
total := 0;
número := 1;
WHILE número >= 0 DO
BEGIN
  read (número);
  total := total + número
END
```

En seguida se probará a mano la solución propuesta para verificar si es correcta. Supóngase que los datos de entrada consisten en un solo número positivo y un centinela:

100 —999

El ciclo anterior no funciona correctamente. El valor de *total* debería ser 100 después de terminar el ciclo. Sin embargo, el valor real, como puede comprobar el lector, es  $-999 + 100$ , o sea  $-899$ .

El problema se puede resolver de manera correcta con la técnica llamada *preparación*. Es posible asignar un valor inicial a *número* antes de entrar al ciclo si se lee el primer valor. Este valor se procesará dentro del ciclo. La última proposición del ciclo leerá el siguiente valor que se va a procesar, ya sea como dato o como centinela. El siguiente segmento de código resuelve correctamente el problema. Se recomienda al lector verificar que es correcto con los valores 100 y —999, así como otros valores de prueba. Por ejemplo, ¿por qué podría ser útil como caso de prueba el valor único —999?

```
total:=0
read (número); (* asignar valor inicial a número *)
WHILE número > = 0 DO (* todavía no se lee el centinela *)
BEGIN
  total := total + número; (* sumar un dato no centinela *)
  read (número) (* capturar el siguiente dato o centinela *)
END
```

Se ha ilustrado una vez más la importancia de probar las soluciones a través de grupos pequeños de datos. Casi todas las dificultades que presentan las soluciones de los problemas de programación se pueden explicar si se ejecutan a mano, y cuidadosamente, segmentos pequeños del programa *antes de utilizar el código en un segmento de programa mayor*. Esta técnica no es muy recomendable.

He aquí una lista de recordatorios importantes que se aplican al uso de las construcciones cíclicas del Pascal.

- Al probar a mano el código se pueden identificar prácticamente todos los errores presentes en las soluciones de los problemas.
- La expresión booleana de la proposición WHILE se debe cumplir para que el ciclo se ejecute.
- La expresión booleana de la proposición REPEAT-UNTIL debe ser falsa para que el ciclo se repita.
- El ciclo REPEAT-UNTIL siempre se ejecuta por lo menos una vez.
- En una proposición WHILE o REPEAT-UNTIL, una variable de conteo lleva la cuenta del número de veces que se ha ejecutado el cuerpo del ciclo.
- Se pueden emplear variables booleanas conocidas como indicadores para controlar la ejecución de una proposición WHILE o REPEAT-UNTIL.
- Si aparece más de una proposición en el cuerpo de una proposición WHILE o FOR, las proposiciones deben estar delimitadas por medio de una pareja BEGIN-END.
- El valor de la variable de conteo de un ciclo FOR queda sin definir al completarse el ciclo.

## SECCIÓN 6.7 REVISIÓN DEL CAPÍTULO

En este capítulo se analizaron los ciclos y las estructuras de control cíclicas correspondientes en Pascal. En particular, se habló de las tres estructuras de control de ciclos WHILE, REPEAT-UNTIL y FOR. La proposición WHILE es un ciclo que se ejecuta en tanto se cumpla la expresión booleana especificada. Cuando se vuelve falsa la expresión, se pasa por alto el cuerpo del ciclo y se ejecuta la proposición que sigue al ciclo. También se pueden utilizar centinelas e indicadores para controlar el ciclo.

La proposición REPEAT-UNTIL es una estructura de control de ciclos que se ejecuta por lo menos una vez, y después se prueba una expresión booleana. Si la expresión booleana es falsa, el ciclo se repite. Cuando se cumple la expresión el ciclo termina, y la ejecución continúa con la siguiente proposición después de la proposición REPEAT-UNTIL. Las proposiciones FOR se pueden usar cuando se conoce el número de veces que se va a ejecutar el ciclo (ya sea como valor de una expresión o como una constante) antes de entrar al ciclo. La proposición FOR es una estructura de control definida.

En este capítulo se compararon las tres estructuras de control de ciclos y se analizaron las condiciones que son favorables para su uso. También se presentó la resolución de problemas mediante ciclos con un ejemplo detallado de proceso de textos. También se presentó un resumen de las estructuras de control fundamentales. A continuación se ofrece un resumen de las construcciones cíclicas de Pascal que puede servir como referencia en el futuro.

## REFERENCIAS DE PASCAL

- 1 Proposición WHILE: ejecuta el cuerpo del ciclo mientras (*while*) se cumple la condición.

**WHILE expresión-boleana DO  
proposición**

Ejemplo:

```
WHILE (número > 0) DO
BEGIN
    cuenta := cuenta + 1;
    read (número)
END
```

- 2 Proposición REPEAT-UNTIL: repite (*repeat*) el cuerpo del ciclo hasta que (*until*) se cumple la condición.

```
REPEAT
    proposición
    proposición
    ...
    proposición
UNTIL expresión-boleana
```

Ejemplo:

```
REPEAT
    read (número);
    cuenta := cuenta + 1
UNTIL (número > 100)
```

- 3 Proposición FOR: ejecuta el ciclo desde el valor inicial hasta el valor final.

**FOR** cuenta := valor-inicial **TO** valor-final **DO** proposición;  
**FOR** cuenta := valor inicial **DOWNTO** valor-final **DO** proposición;

Ejemplo:

```
FOR i := 1 TO 10 DO
BEGIN
    read (número);
    suma := suma + número
END
```

- 4 Estructuras de control fundamentales

- 4.1 Ejecución secuencial
- 4.2 Procedimiento
- 4.3 Selección
- 4.4 Ciclo

Hasta aquí se han presentado las estructuras de control fundamentales en Pascal. Ahora se estudiarán los procedimientos, funciones y parámetros. En particular, se examinarán procedimientos anidados, funciones estándar en Pascal, funciones definidas por el usuario y funciones recursivas.

**Palabras clave del capítulo 6**

aserción	proposición FOR
centinela	proposición GOTO
ciclo infinito	proposición REPEAT-UNTIL
ciclos	proposición WHILE
estructura de control definida	transferencia incondicional
estructura de control indefinida	umbral
indicador	variable de control de ciclo
invariable de ciclo	variante de ciclo
poscondición	verificación de programas
precondición	

**EJERCICIOS DEL CAPÍTULO 6****★ EJERCICIOS ESENCIALES**

- 1 Escríbase un programa que exhiba la tabla de multiplicación desde uno por uno hasta nueve por nueve. Deberá comenzar aproximadamente así:

1	2	3	
1	1	2	3
2	2	4	6

- 2 Escríbase de nuevo el código

**REPEAT**  
    **cuerpo**  
**UNTIL expresión-booleana**

con un solo ciclo WHILE.

- 3 Los botones de un teléfono están organizados según este patrón:

1	2	3
4	5	6
7	8	9
*	0	#

Los datos de entrada incluirán siete enteros, todos en la escala de cero a nueve, que represente un número telefónico (sin incluir \* o #). Escríbese un programa en Pascal que exhiba los números de renglón y columna que se deben oprimir para marcar el número.

Ejemplo de entrada:

5 6 7 8 0 1 3

Ejemplo de salida:

2	2
2	3
3	1
3	2
4	2
1	1
1	3

- 4 Supóngase que los datos de entrada constan de renglones, cada uno de los cuales contiene una letra en la primera columna, seguida de un valor real que representa un importe en dólares. El último renglón contiene únicamente la letra *X* en la columna uno. El primer renglón contiene la letra *A* y el saldo anterior de una cuenta de cheques. Los demás renglones contienen la letra *D* y el importe de un depósito o la letra *R* y el importe de un retiro. Determíñese el saldo exacto de la cuenta después de procesar las transacciones.

Ejemplo de entrada

P 1200.35  
D 64.12  
W 390.00  
W 289.67  
D 13.02  
W 51.07  
X

Ejemplo de salida:

El saldo final es 546.75.

## ★ ★ EJERCICIOS IMPORTANTES

- 5 Considérese la función

$$f(x,y,z) = x^2 + 2xy - 3yz + xz$$

donde *x*, *y* y *z* son valores enteros. Encuéntrese el valor más alto de la función para valores de *x*, *y* y *z* en la escala de -3 a 3. Exhíbanse los valores correspondientes de *x*, *y*, *z* y *f(x,y,z)*.

- 6 Supóngase que el Pascal permitiera el uso de variables reales como variables de control en ciclos FOR de la forma

257

CICLOS

FOR r := inicio TO fin BY incremento

donde  $r$ , *inicio*, *fin* e *incremento* son todas del tipo real. ¿Qué problemas puede detectar el lector en este ciclo? En particular, ¿cómo afectaría la naturaleza aproximada del tipo de datos real la propiedad que tienen los ciclos FOR de ejecutar siempre el cuerpo de un ciclo un número definido de veces?

- 7 Los datos de entrada para este problema son el importe de un depósito periódico a una cuenta de ahorros (como número real) y el número de períodos de depósito. Determinese el interés compuesto para cada una de las siguientes tasas de interés: 8, 8.5, 9, 9.5 y 10%.

### ★★★ EJERCICIOS ESTIMULANTES

- 8 En los libros de juegos y acertijos se encuentran muchos problemas similares al siguiente. El diagrama que se presenta en seguida muestra las posiciones de los dígitos decimales en un problema de multiplicación. También A, B y C representan tres dígitos decimales únicos y cada \* representa algún dígito decimal. Dedúzcase un esquema de búsqueda apropiada para encontrar los valores de A, B y C.

$$\begin{array}{r} & A & B & C \\ & B & A & C \\ \hline & * & * & * & * \\ & * & . & * & \\ * & * & * & B \\ \hline & * & * & * & * & * & * \end{array}$$

## PROBLEMAS DEL CAPÍTULO 6 PARA RESOLUCIÓN EN COMPUTADORA

### ★ PROBLEMAS ESENCIALES

- 1 Los datos de entrada incluyen uno o más números reales positivos con un número negativo al final (el centinela). Determinese la desviación estándar de estos datos (sin incluir al centinela). La desviación estándar de un conjunto de números  $x_1, x_2, \dots, x_n$  se define como la raíz cuadrada de la expresión  $(s/n) - a^2$ , donde  $a$  es el promedio de los valores  $x$  (es decir,  $(x_1 + x_2 + \dots + x_n)/n$ ) y  $s$  es la suma de los cuadrados de los valores  $x$  (es decir,  $x_1^2 + x_2^2 + \dots + x_n^2$ ). Exhíbase el resultado como se muestra en el ejemplo, con dos dígitos a cada lado del punto decimal.

Ejemplo de entrada:

25.0    23.0    22.0    21.0    17.0  
9.0    6.0    5.0    -1.0

Ejemplo de salida:

**La desviación estándar es 7.60**

- 2 Dado un renglón de datos de entrada que contiene 20 dígitos decimales sin espacios, conviértanse en cinco enteros de cuatro dígitos cada uno.
- 3 Dado un fragmento de texto terminado por el centinela \$, determinese el nombre de consonantes y vocales dobles. Por ejemplo, el texto “Llama al chico que lee” tiene dos consonantes dobles (Ll y ch) y una vocal doble (ee).
- 4 Determinese y exhibase el número más grande y el más pequeño en un conjunto de enteros capturado que termina con el número –9999.
- 5 Exhíbanse todos los números primos entre A y B, donde A y B son enteros positivos capturados de los datos de entrada. Un número primo es cualquier entero positivo mayor que uno y que es divisible sólo entre sí mismo y la unidad. Una forma sencilla de determinar si un número dado  $N$  es número primo es intentar la división entre dos y entre todos los números nones mayores que uno y menores que la raíz cuadrada de  $N$ . Si cualquiera de estos números divide a  $N$  en forma exacta, no se trata de un número primo.
- 6 Cada uno de los renglones de datos de entrada contiene el nombre de una persona y su dirección, con cada “renglón” de la dirección separado por una línea vertical (|), y con un signo de dólares al final. El último renglón contiene únicamente un signo de dólares. Exhíbanse los datos como si se fuera a imprimir en etiquetas de correo. Se supone que cada etiqueta tiene cinco renglones. Por ejemplo, si los datos de entrada fueran

John Smith|32 Tenth Avenue North|Bellvue, Wa. 60123\$  
 Bryan Meeks|1201 E. 89th #21|Nowhere, Ontario\$  
 \$

la salida sería

John Smith  
 32 Tenth Avenue North  
 Bellvue, Wa. 60123

Bryan Meeks  
 1201 E. 89th #21  
 Nowhere, Ontario

- 7 Utilícese una serie infinita para determinar el valor de  $e^x$ . Úsese la fórmula

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$$

Supóngase que los datos de entrada contienen un valor positivo de  $x$  y un valor positivo  $eps$ . Calcúlense y súmense términos a la suma en tanto el valor absoluto de cada término sea mayor que  $eps$ .

- 8 Los datos de entrada son una lista de enteros que supuestamente está en orden ascendente. El último entero de la lista es 999. Léase la lista y, si está correcta, exhibase el mensaje “CORRECTO”. En caso contrario, imprímanse los dos números que preceden al valor erróneo (si hay dos; de otro modo, imprímase solamente el anterior) y el valor erróneo. Por ejemplo, si los datos de entrada fueran

**5 7 12 31 95 89 112 999**

la salida sería

**31 95 89.**

Si la entrada fuera

**5 7 12 31 95 98 112 999**

la salida sería

**CORRECTO.**

Por último, si la entrada fuera

**53 45 59 999**

la salida sería

**53 45.**

### ★ ★ PROBLEMAS IMPORTANTES

- 9 Las computadoras pueden codificar fácilmente los mensajes. Dado un entero  $n$  en la primera línea de los datos de entrada, efectúese una “rotación” del mensaje que se encuentra en la segunda línea y termina con \$ por  $n$  caracteres y exhibase el resultado. El mensaje incluirá únicamente letras mayúsculas y espacios en blanco. La rotación de un carácter alfabético por  $n$  caracteres se puede ilustrar de la siguiente manera. Escríbase primero el alfabeto dos veces en dos renglones. En seguida desplácese todos los caracteres del segundo renglón  $n$  caracteres hacia la izquierda. Los que “sobren” del lado izquierdo se colocan en el extremo derecho. Por ejemplo, si  $n$  es tres, se obtiene la siguiente figura:

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	D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
---	---

El mensaje “ENVÍEN A LA CABALLERÍA\$” quedaría cifrado para producir el mensaje “HQYLNQ D OD FDEDOOHULD”.

- 10 Repítase el problema 9 pero en este caso hágase la transformación inversa. Es decir, dado un mensaje codificado, hágase la “rotación” inversa para producir texto legible.
- 11 Determinese el valor de *pi* (3.14159...) mediante la estimación del área de un cuadrante de círculo. El área del cuadrante se puede determinar si se cuenta el número de “dardos” que caen dentro del cuadrante de un círculo trazado de un cuadrado cuyo lado es igual al radio del círculo. Para determinar la posición del dardo se generan dos números aleatorios en la escala de 0.0 al radio del círculo y se emplean estos números aleatorios como coordenadas de la posición en la que quedó el dardo. Casi todos los sistemas de cómputo incluyen una función externa para calcular estos números aleatorios. (Véase el Cap. 7.)

### ★★★ PROBLEMAS ESTIMULANTES

- 12 Imprímanse todos los primos gemelos entre *A* y *B*. Los primos gemelos son parejas de números primos con una diferencia entre sí de exactamente dos. Por ejemplo, tres y cinco son primos gemelos, al igual que 11 y 13 y 17 y 19.
- 13 Supóngase que el número más grande de dígitos decimales que se pueden almacenar en una variable entera es de cuatro. Por tanto, el valor entero más alto que se puede almacenar será 9999. Los datos de entrada constan de 10 enteros de 20 dígitos que se deben sumar para producir un resultado con un máximo de 22 dígitos. Calcúlese y exhibase el resultado. (Sugerencia: obsérvese que el resultado máximo de sumar dos números de cuatro dígitos es de 19,998, que se puede manejar como la suma de 9998 y un acarreo de uno.)
- 14 Los datos de entrada contienen una lista de nombres, cada uno encerrado entre apóstrofos y no mayor de 20 caracteres, y una calificación para cada persona. Se usará el centinela \$ para marcar el fin de los datos. Exhibase cada nombre en las columnas de la 1 a la 20 de un renglón de salida y la calificación correspondiente (en la escala de cero a 100) en las columnas 23 a 25 del mismo renglón. Por último, exhibase la palabra PROMEDIO en las columnas uno a ocho y la calificación promedio (redondeada al entero más cercano) en las columnas 23 a 25 del último renglón.

Ejemplo de entrada:

```
'CARLOS G.'    73  'MARÍA B.'   100  'PABLO C.'   86
'SUSANA F.'    91  'BRUNO M.'   100  $
```

Ejemplo de salida:

CARLOS G.	73
MARÍA B.	100

- 15 Un grupo de documentos se caracteriza por tener uno o más de entre siete atributos. Cada documento se especifica mediante un solo renglón en el que se indica la presencia o ausencia de los atributos mediante una *X* o un espacio en blanco en las primeras siete columnas nubes (1, 3, 5, 7, 9, 11 y 13) y el nombre del documento seguido de un punto, a partir de la columna 15. El primer renglón de datos de entrada tendrá una “máscara” de atributos en las primeras 13 columnas, seguida de un entero que representa el número de atributos que deben coincidir con un documento para que se pida ese documento. El último renglón de los datos de entrada tendrá el signo \$ en la columna 1. Prodúzcase una lista de los documentos que corresponda a la solicitud en una forma similar a la que se muestra en el ejemplo.

Ejemplo de entrada

```

X X X X X X X 4
X X     X Pascal para todos.
X X X X X X X Introducción a la computación con Pascal.
X X     X El Pascal difícil.
X X     X X Hombres, máquinas y Pascal.
$
```

Ejemplo de salida:

```

Introducción a la computación con Pascal.
El Pascal difícil.
Hombres, máquinas y Pascal.
```

- 16 Se coloca un sensor para medir el volumen de tránsito en una calle de un solo carril. Cada vez que un par de ruedas pasa sobre el sensor se produce un renglón de salida con el tiempo (en forma de un número real de segundos). La longitud máxima esperada de los vehículos es de 18 metros y la velocidad máxima esperada es de 120 kilómetros por hora (se supone que las personas exceden el límite de velocidad). Sin embargo, deberá ser posible cambiar fácilmente estos valores (es decir, empléense constantes para ellos). Todas las señales que se detecten dentro del periodo permitido para el paso de un vehículo contarán como un vehículo. Determinese el número de vehículos que pasaron durante el periodo de medición, así como el número promedio de vehículos por minuto. Como problema adicional (más difícil), determinense las velocidades máxima y mínima de los vehículos (suponiendo que todos los vehículos tienen por lo menos dos ejes).
- 17 Realicense las funciones de una calculadora simple. Los datos de entrada serán una secuencia de dígitos decimales y los operadores +, -, \* y /, seguida

de un signo de igual. Hágase caso omiso de los espacios en blanco. Los operadores se aplican en el orden en que aparecen en los datos de entrada, y producen resultados enteros.

Ejemplo de entrada

$4 + 3 / 2 * 8 - 4 =$

Ejemplo de salida:

20

- 18 Repítase el Prob. de la calculadora (problema 17), pero además de dígitos decimales y operadores, los datos pueden contener también  $A n$  después de un dígito decimal o  $R n$  en vez de un dígito decimal, donde  $n$  es 0, 1, 2, o 3. Éstos representan el almacenamiento del resultado de un cálculo en un “registro” o la recuperación de un resultado previo. La recuperación de un resultado que no ha sido almacenado deberá producir un error.

Ejemplo de entrada:

$4 + 3 A 0 / 2 + R 0 =$

$4 + 3 A 0 / 2 + R 1 =$

Ejemplo de salida:

10

ERROR

- 19 Dado un entero  $n$  (como dato de entrada), imprímanse las líneas 1 a  $n$  del triángulo de Pascal. La línea  $i$  del triángulo de Pascal contiene los coeficientes de la expansión polinomial de  $(x + 1)^{i-1}$ . Por ejemplo, para un valor de entrada de cinco la salida sería la siguiente:

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

- 20 Imprímase una tabla de todas las temperaturas enteras desde  $A$  hasta  $B$  grados Celsius incluyendo las temperaturas Fahrenheit enteras. La temperatura Fahrenheit que corresponde a  $C$  grados Celsius está dada por la expresión

$$32 + 9 * C / 5$$

Por ejemplo, si  $A = 0$  y  $B = 5$ , la salida deberá verse así:

Celsius	Fahrenheit
0	32
	33
1	
	34
	35
2	
	36
	37
3	
	38
	39
4	
	40
5	
	41

- 21 Se da una secuencia de 10 valores enteros, todos menores de 60. Prodúzcase una gráfica de barras horizontales similar a la que se muestra en el ejemplo para estos datos.

Ejemplo de entrada:

5 12 17 35 52 42 12 28 31 8

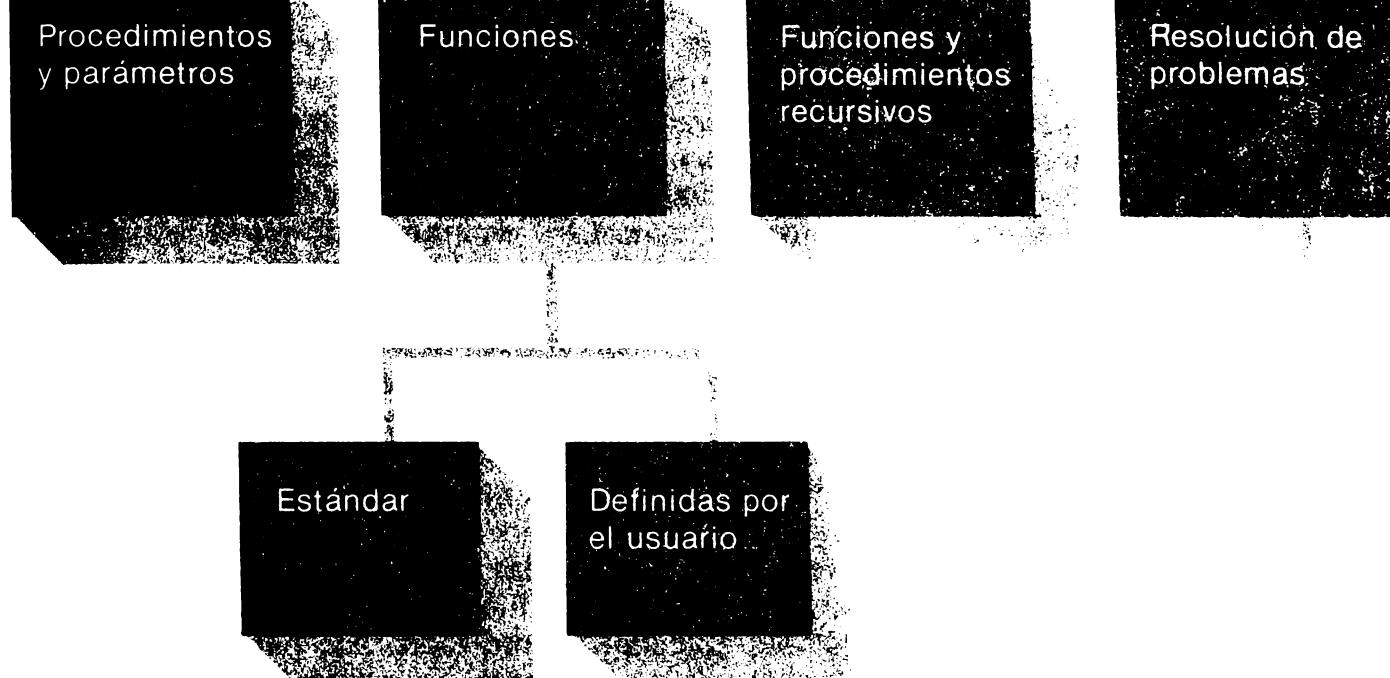
Ejemplo de salida:

```
*****  
5 *****  
*****  
*  
*****  
12 *****  
*****  
*  
*****  
17 *****  
*****  
*  
*****  
35 *****  
*****  
*  
*****  
52 *****  
*****  
*  
*****  
42 *****  
*****  
*  
*****  
12 *****  
*****  
*  
*****
```

```
28 ****  
*****  
*  
*****  
31 ****  
*****  
*  
*****  
8 ****  
*****
```

# CAPÍTULO 7

## CAPÍTULO 7 PROCEDIMIENTOS Y FUNCIONES



## PROCEDIMIENTOS Y FUNCIONES

## OBJETIVOS

Después de completar este capítulo, el lector deberá ser capaz de:

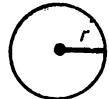
- Reconocer y aplicar parámetros de valor y variables en procedimientos y funciones
- Distinguir entre parámetros formales y verdaderos
- Escribir procedimientos anidados
- Determinar el alcance de los procedimientos, funciones e identificadores locales y globales
- Reconocer y aplicar las funciones booleanas estándar: *odd*, *eoln* y *eof*
- Escribir funciones definidas por el usuario, incluso funciones recursivas
- Reconocer y aplicar funciones generadoras de números aleatorios
- Resolver, probar y depurar problemas que emplean procedimientos y funciones

## PANORAMA GENERAL DEL CAPÍTULO

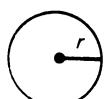
Este capítulo contiene abundante información sobre procedimientos, funciones, y sus aplicaciones. Se comienza por repasar los parámetros de procedimientos. En particular, se analizan los parámetros de valor y variables y la diferencia entre parámetros formales y verdaderos. En seguida se presenta un análisis de los procedimientos anidados (es decir, procedimientos dentro de otros procedimientos). Se incluyen algunos ejemplos y se pone especial atención en las declaraciones locales que puede tener cada procedimiento. Se examina específicamente el alcance de los identificadores locales. En otras palabras, se estudia la parte del programa en la que se conoce el identificador. Se analizan los identificadores locales y globales en relación con los procedimientos anidados.

En la sección 7.2 se analizan las funciones, tanto estándar como definidas por el usuario. Las funciones estándar que se verán son las funciones booleanas: *odd* (probar entero para determinar si es non), *eoln* (probar si es fin de línea) y *eof* (probar si es fin de archivo). Después de esto se presenta un análisis de las funciones definidas por el usuario. También se incluyen aplicaciones de las funciones. En Pascal, una función puede llamarse a sí misma. Una función así se llama **recursiva**. Se presentará un ejemplo cuidadosamente elaborado para que el lector comprenda la naturaleza de las funciones recursivas. También se incluyen varios ejemplos más.

Al final del capítulo se aplicarán procedimientos y funciones a un problema referente a un programa de juego para multiplicar números. En la sección 7.4 se analizarán los generadores de números aleatorios y la forma como se utiliza una función así en este problema. Por último, se termina el capítulo con un análisis de efectos secundarios, cambios en el orden de las declaraciones de procedimientos y funciones y transportabilidad.



Área (región sombreada) =  $\pi \cdot r^2$



Circunferencia (perímetro) =  $2 \cdot \pi \cdot r$

Figura 7.1 Área y circunferencia de un círculo.

## SECCIÓN 7.1 REPASO DE PROCEDIMIENTOS Y PARÁMETROS

En esta sección se examinarán los procedimientos y parámetros con mayor detalle y se aplicará el resultado más tarde a las funciones. Se verá una vez más lo que se ha aprendido con un ejemplo. Supóngase que se debe escribir un procedimiento en Pascal que calcule el área y circunferencia de un círculo de radio  $r$  (véase la Fig. 7-1). Recuérdense las siguientes dos fórmulas para los círculos, donde  $\pi$  es una constante que vale aproximadamente 3.14159:

$$A = \pi \cdot r^2 \quad (\text{área} = \pi \text{ por radio al cuadrado})$$

$$C = 2 \cdot \pi \cdot r \quad (\text{circunferencia} = 2 \text{ por } \pi \text{ por radio})$$

Recuérdese que un procedimiento es un subprograma que puede tener parámetros de entrada (de valor) y de salida (variables). ¿Cuáles son los parámetros de entrada (de valor)? Es decir, ¿qué información debe enviarse como dato de entrada al procedimiento? En este caso, el radio  $r$  es el único parámetro de entrada (de valor). La constante  $\pi$  (aproximadamente 3.14159) se puede declarar dentro del procedimiento. La salida de este procedimiento será el área y circunferencia del círculo. Así, *área* y *circunferencia* serán los parámetros de salida (variables) del procedimiento. La tarea se puede realizar mediante el siguiente procedimiento llamado *círculo*:

PROCEDURE círculo (radio: real; VAR área, circun: real);

(\* Procedimiento para calcular el área y \*)

(\* circunferencia de un círculo, dado su radio \*)

CONST

pi = 3.14159;

BEGIN

área: pi \* sqr (radio);

circun: = 2 \* pi \* radio

END;

Nótese que se usó la función estándar (incluida) *sqr* del Pascal para obtener el cuadrado del radio en el cálculo del área. Recuérdese que la palabra reservada VAR no precede al parámetro de entrada (de valor) *radio*.

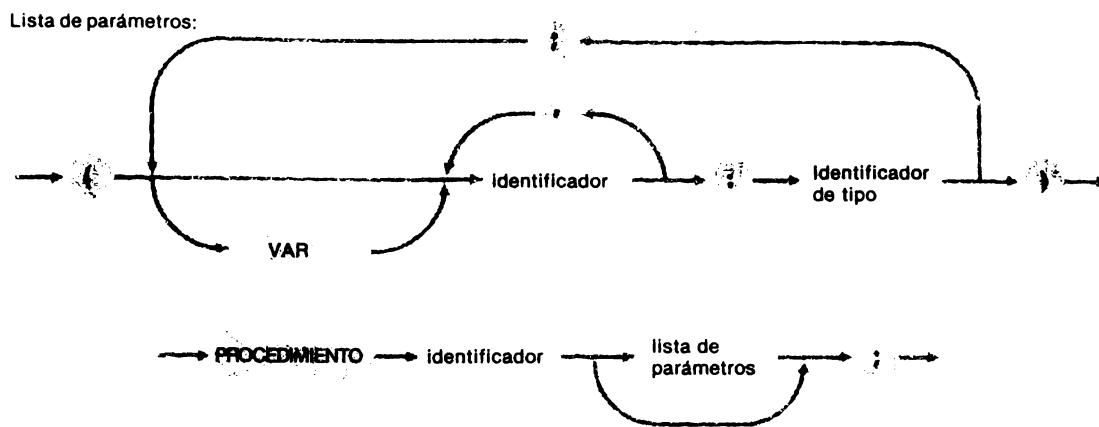


Figura 7-2 Diagrama de sintaxis para lista de parámetros y procedimiento.

### Parámetros de valor y variables

El encabezado del procedimiento contiene el nombre del procedimiento y una lista opcional de nombres de variables con tipos de datos. Los nombres de variables en el encabezado del procedimiento se llaman *parámetros formales*, ya que no se conocen los valores verdaderos en tanto no se llame o invoque al procedimiento. En la figura 7-2 se muestran los diagramas de sintaxis de las listas de parámetros y de los encabezados de procedimientos.

Obsérvese que todos los parámetros formales deben incluir un tipo de datos en el encabezado y que algunos parámetros van precedidos de la palabra reservada VAR. Cierta información, como *radio* en el procedimiento *círculo*, se proporciona al procedimiento como datos de entrada y no debe cambiarse en el progra-

Figura 7-3 Invocación de procedimientos.

```

PROCEDURE círculo (radio : real; VAR área, circun : real);
CONST
  pi = 3.14159;
BEGIN
  área := pi * sqr(radio);
  circun := 2 * pi * radio
END;
(* programa principal *)
BEGIN
  .
  .
  (* invocación del procedimiento *)
  círculo (5, área, circunferencia)
  .
  .
END.

```

ma que llama. Los parámetros formales que corresponden a los datos de entrada de un procedimiento se llaman parámetros de entrada (de valor). En Pascal se le llama sencillamente *parámetro de valor* y en adelante se usará esta terminología. En el encabezado del procedimiento, la lista de parámetros de valor *no* incluye la palabra reservada VAR.

Los parámetros formales que corresponden a la salida de procedimiento, como son el área y la circunferencia del círculo, se conocen como parámetros de salida (variables). En Pascal se les llama sencillamente *parámetros de variables*. Cuando se escribe la lista de estos parámetros es preciso emplear la palabra reservada VAR. Esto garantiza que los parámetros verdaderos en el programa que llama van a cambiar. Por ejemplo, si se invoca el procedimiento *círculo* mediante la siguiente proposición que incluye los parámetros formales

### **círculo (5, área, circunferencia)**

el procedimiento modificará el contenido de los parámetros variables *área* y *circunferencia*. El parámetro de valor 5 no cambiará. Cuando se invoca un procedimiento, el orden y tipo de datos de los parámetros verdaderos deben concordar con el orden y tipo de datos de los parámetros formales (véase la Fig. 7-3).

El parámetro verdadero que corresponde a un parámetro de valor puede ser una variable, una constante o una expresión. Cuando se invoca el procedimiento, se obtiene el valor del parámetro de valor y se pasa al procedimiento. Por ejemplo, la siguiente invocación de procedimiento pasa una expresión aritmética como parámetro de valor verdadero:

### **círculo (2 \* 3 - 1, área, circunferencia)**

Cuando se invoca el procedimiento, el parámetro de valor formal *radio* tendrá el valor calculado de cinco. Por otro lado, el parámetro verdadero que corresponde a un parámetro variable debe ser una variable. Por tanto, la invocación del procedimiento

### **círculo (5, 2, 1)**

*no* es válida, ya que los parámetros segundo y tercero son parámetros de salida o variables y 2 y 1 no son variables.

Se ha hablado de los parámetros como datos de entrada o salida de un procedimiento. Sin embargo, algunos parámetros pueden funcionar como entrada y salida simultáneamente. En Pascal, a éstos se les llama también parámetros variables y deben incluir la palabra reservada VAR en la lista de parámetros formales del encabezado del procedimiento. Ahora se verá un ejemplo en el que se usan parámetros variables como entrada y también como salida. Considérese el problema de escribir un procedimiento llamado *com muta* que comuta, o intercambia, el contenido de dos localidades de memoria llamadas *memoria1* y *memoria2*. Si se estudia con detenimiento este problema se verá que no se puede intercambiar el contenido de las localidades directamente de esta manera:

```
memoria1 := memoria2;
memoria2 := memoria1;
```

Esta secuencia de proposiciones no funcionará, ya que la primera proposición de asignación destruirá el contenido de *memoria1*. Es preciso guardar temporalmente el contenido de *memoria1*. En este procedimiento *memoria1* y *memoria2* funcionan como entrada y como salida, ya que el contenido de ambas localidades va a cambiar.

```
PROCEDURE conmuta (VAR memoria1, memoria2: integer);
(* Intercambia el contenido de memoria1 y memoria2 *);
VAR
    temp : integer;          (* localidad temporal *)
BEGIN
    temp := memoria1;      (* guardar memoria1 *)
    memoria1 := memoria2;  (* cambiar memoria2 *)
    memoria2 := temp;      (* cambiar memoria1 *)
END;
```

Al escribir procedimientos, ¿cómo se decide si se deben emplear parámetros de valor o variables? En general, si el procedimiento envía información de vuelta al programa que lo llamó, se deben usar parámetros variables. Por otro lado, si la información se pasa al procedimiento únicamente como datos de entrada, deben usarse parámetros de valor. Existen algunas excepciones a esta regla, que se examinarán en capítulos posteriores. Pero, ¿por qué no usar sólo parámetros variables? Básicamente, la razón es que cuando se usa un parámetro de valor, se hace una copia del valor y el procedimiento emplea esta copia. Esto garantiza que el procedimiento no puede cambiar el parámetro de valor original. No obstante, los parámetros de valor pueden ser costosos. Como se verá más adelante, es posible que se quiera pasar como datos de entrada a un procedimiento un conjunto de muchos valores (como una tabla de nombres o un archivo de caracteres). Aun cuando estos valores no funcionen sino como datos de entrada, normalmente conviene declararlos como parámetros variables. Si no se hace así, la computadora gastará tiempo y localidades de memoria al copiar el conjunto de valores. Debe tenerse cuidado de que el procedimiento no cambie inadvertidamente estos valores.

He aquí un resumen de algunos puntos importantes en lo que concierne a los parámetros de valor y variables.

- Se debe incluir una lista de los parámetros formales en el encabezado del procedimiento.
- La proposición de invocación del procedimiento pasa los parámetros verdaderos.

- El orden y tipo de datos de los parámetros formales deben ser los mismos que para los parámetros verdaderos.
- Los parámetros variables deben incluir la palabra reservada VAR en la lista de parámetros de un encabezado de procedimiento.
- El parámetro verdadero de un parámetro variable debe ser una variable.
- El parámetro verdadero de un parámetro de valor puede ser una variable, constante o expresión.

## Procedimientos anidados

Cuando se escriben procedimientos dentro de otros procedimientos se dice que están *anidados*. En seguida se verá un ejemplo.

### Problema 7.1

*Supóngase que se desea escribir un procedimiento llamado buscar que lee una cadena de caracteres que contiene un nombre seguido de un salario representando por un entero con un signo de dólares. El objetivo de este procedimiento es exhibir el salario si rebasa cierto límite; en caso contrario se debe hacer al salario igual a cero. Una cadena de datos de entrada representativa se vería así:*

JUAN PÉREZ \$50000

El procedimiento *buscar* incluirá dos procedimientos llamados *dólar* y *sueldo*. El procedimiento *dólar* leerá caracteres hasta que se encuentre un signo de dólares, en tanto que el procedimiento *sueldo* leerá el entero *salario* que sigue al signo de dólar y determinará si rebasa el límite. El procedimiento *buscar* exhibirá un mensaje si el salario rebasa el límite o devolverá un valor de cero al programa principal si el salario no rebasa el límite.

Primero se escribirá el procedimiento *dólar* para leer y pasar por alto caracteres hasta inmediatamente después de leer el primer signo de dólares.

```

PROCEDURE dólar
(* Lee hasta encontrar $ *)
VAR
  uncar: char;
BEGIN
  REPEAT
    read (uncar)
  UNTIL (uncar = '$')
END;
```

Obsérvese que se declaró una variable de carácter llamada *uncar* dentro del procedimiento para almacenar el carácter leído. Esta variable es una *variable local* que se conoce únicamente en el procedimiento *dólar*. Este procedimiento no tiene parámetros formales porque ya se sabe que el último carácter leído será un signo de dólar.

En seguida se escribirá el procedimiento *sueldo* para leer el salario (entero) que sigue al signo de dólares. Puesto que se desea pasar el salario al procedimiento *buscar*, se debe declarar como parámetro variable. También se incluye una variable booleana para indicar si el salario rebasa el límite (que es un parámetro de valor).

```
PROCEDURE sueldo (límite: integer; VAR salario: integer;
                   VAR indic: Boolean);
(* Lee el salario y determina si rebasa el límite *)
BEGIN
```

```
    read (salario);
    indic := salario > límite
```

```
END;
```

Obsérvese que en el encabezado del procedimiento aparece dos veces la palabra VAR. Si se omitiera el último VAR (antes de *indic*), *indic* sería un parámetro de valor y la proposición de asignación

```
indic := salario > límite
```

no cambiaría al parámetro verdadero. Cabe hacer notar, empero, que en este caso *indic* cambiaría localmente.

Ahora ya se puede escribir el procedimiento principal *buscar* que tiene un parámetro de valor llamado *límite* y un parámetro variable llamado *paga*.

```
PROCEDURE busca (límite: integer; VAR paga: integer);
(*Procedimiento para determinar si el salario rebasa el límite *)
VAR prueba: Boolean;
```

```
PROCEDURE dólar;
(* Lee hasta encontrar $ *)
VAR
  uncar: char;
BEGIN
```

```
REPEAT  
    read (uncar)  
UNTIL (uncar = '$')
```

```
END;
```

```
PROCEDURE sueldo (límite: integer; VAR salario: integer;  
                  VAR indic: Boolean);
```

(\* Lee el salario y determina si rebasa el límite \*)

```
BEGIN
```

```
    reand (salario);  
    indic := salario > límite
```

```
END;
```

(\* Aquí comienza el procedimiento buscar \*)

```
BEGIN
```

```
    dólar;                      (* buscar $ *)  
    salario (límite, paga, prueba);  (* leer salario *)  
    IF prueba  
        THEN writeln ('El salario rebasa el límite:', paga)  
    ELSE paga := 0
```

```
END;
```

## Reglas de alcance

Cuando se tienen procedimientos anidados, el programador debe ocuparse de variables locales y globales y su alcance. El *alcance* de un identificador es aquella parte del programa en la que se conoce el identificador. Para ver esto, examíñese minuciosamente el ejemplo anterior del procedimiento *buscar*. El alcance de la variable local *uncar* en el procedimiento *dólar* es solamente el procedimiento *dólar*. La variable *uncar* no se conoce en el programa principal ni en el procedimiento *suelo*. Por otro lado, la variable *prueba* en el procedimiento *buscar* es una variable global que se conoce en los procedimientos *dólar* y *suelo*. Puesto que cualquier procedimiento puede incluir declaraciones CONST y VAR, los identificadores se conocen como constantes locales o variables locales en el procedimiento. Cada uno de los identificadores se conoce en todos los segmentos del procedi-

miento, incluso en los procedimientos anidados que no tienen un identificador con el mismo nombre. Por ejemplo, considérese la siguiente estructura de procedimientos anidados (obsérvense los bloques de cada procedimiento).

```
PROCEDURE externo (x: integer; VAR y: integer);  
CONST
```

```
    t := 60;
```

```
VAR
```

```
    a, b, c: integer;
```

```
PROCEDURE interno1 (x, y: integer);  
VAR
```

```
    a, z: integer;
```

```
BEGIN
```

```
END; (* interno1 *)
```

```
PROCEDURE interno2 (VAR a: integer);  
VAR
```

```
    v, w: integer;
```

```
PROCEDURE másinterno (VAR x, y: integer);  
VAR
```

```
    a: integer;
```

```
BEGIN
```

```
.
```

```
.
```

```
.
```

```
END; (* másinterno *)
```

```
BEGIN  
. . .  
END; (* interno2 *)
```

```
BEGIN  
. . .  
END; (* externo *)
```

Se han dibujado cuadros alrededor de los procedimientos para ayudar al lector a ver el alcance de los identificadores. En primer lugar, la constantes  $t = 60$  y las variables  $a$ ,  $b$  y  $c$  que se declaran en el procedimiento *externo* son identificadores locales del procedimiento externo. No obstante, el alcance de estos identificadores incluye a todo el procedimiento *externo* (cuadro principal). Puesto que los procedimientos *interno1*, *interno2* y *másinterno* están incluidos en el cuadro principal que contiene al procedimiento *externo*, están dentro del alcance de estos identificadores. De hecho, estos identificadores son globales con respecto a los procedimientos *interno1*, *interno2*, y *másinterno*. En otras palabras, la constante  $t = 60$  y las variables  $a$ ,  $b$  y  $c$  en la declaración VAR del procedimiento *externo* se conocen en los procedimientos *interno1*, *interno2* y *másinterno*. Además, los parámetros formales  $x$  y  $y$  del procedimiento *externo* son realmente variables locales del procedimiento *externo* y tienen las mismas reglas de alcance que las declaraciones de constantes y variables del procedimiento *externo*. Por tanto, ellas también son variables globales con respecto a los procedimientos *interno1*, *interno2* y *másinterno*.

Ahora se examinará el procedimiento *interno1*. El alcance de las variables en la declaración es el cuadro que contiene al procedimiento *interno1*. Tómese nota de que el nombre de variable  $a$  aparece tanto en el procedimiento principal (*externo*) como en el procedimiento *interno1*, pero no se refieren a la misma variable. Cualquier referencia al nombre  $a$  en el procedimiento *interno1* será a la variable local  $a$  en el procedimiento *interno1*. *Las variables locales que tienen el mismo nombre que las variables globales tienen prioridad sobre las variables globales*. Para hacer referencia a la variable global, es preciso pasarla como parámetro. Los parámetros del procedimiento *interno1* ( $x$  y  $y$ ) son también variables locales del procedimiento *interno1*.

Por último, se examinará el procedimiento *interno2*. Las variables locales  $v$  y  $w$  de este procedimiento se conocen únicamente dentro del cuadro que contiene al procedimiento *interno2*. Así, las variables  $v$  y  $w$  son realmente variables globales para el procedimiento *másinterno* que está contenido dentro del procedimiento *interno2*. Además, las variables  $a$ ,  $b$  y  $c$  declaradas en el procedimiento principal *externo* son variables globales para el procedimiento *másinterno*. El procedimiento *interno2* tiene una variable local  $a$  (el parámetro formal variable) que también es global con respecto al procedimiento *másinterno*. Sin embargo, el procedi-

miento *másinterno* tiene una variable local con el mismo nombre *a*. Una vez más, en este caso, cualquier referencia a la variable *a* dentro del procedimiento *másinterno* será a la variable *a* declarada localmente dentro del procedimiento *másinterno*.

¿Qué sucede en el caso de las invocaciones de procedimientos? Los nombres de procedimientos tienen alcance, al igual que los nombres de variable. El procedimiento *interno2* puede llamar al procedimiento *interno1*, ya que está más adelante que este último. El procedimiento *interno1* no puede llamar al procedimiento *interno2*. Además, el procedimiento *másinterno* puede llamar al procedimiento *interno1*. Nótese que ningún procedimiento puede invocar al procedimiento *másinterno* fuera del cuadro que contiene al procedimiento *interno2*, ya que no se conoce fuera del procedimiento *interno2*.

En seguida se resumirá lo que se sabe acerca de las reglas de alcance de los identificadores locales y globales.

procedimiento	identificadores conocidos	otras variables o constantes conocidas	procedimientos dentro del alcance
<i>externo</i>	<i>t, a, b, c, x, y</i>	Ninguna	<i>interno1, interno2</i>
<i>interno1</i>	<i>a, z, x, y</i>	<i>t, b, c</i> (en <i>externo</i> )	Ninguno
<i>interno2</i>	<i>v, w, a</i>	<i>t, b, c, x, y</i> (en <i>externo</i> )	<i>másinterno</i>
<i>másinterno</i>	<i>a, x, y</i>	<i>t, b, c</i> (en <i>externo</i> ) <i>v, w</i> (en <i>interno2</i> )	Ninguno

Aun cuando es posible hacer referencia a una variable global desde un procedimiento dentro del alcance de la variable, *generalmente no es buena idea*. Es preferible pasar la variable como parámetro. No conviene cambiar las variables globales mediante una proposición de asignación directa dentro del procedimiento (éste es un ejemplo de *efectos secundario*). Los procedimientos deben ser módulos autosuficientes e independientes. Los efectos secundarios muchas veces pueden provocar problemas serios, como se demuestra en la sección de técnicas de prueba y depuración.

## EJERCICIOS DE LA SECCIÓN 7.1

Los ejercicios 1 a 4 se deben responder de acuerdo al diagrama de bloques que se muestra en seguida.

```
PROGRAM principal (input, output);
VAR a, b : integer;
```

```
PROCEDURE a1 (VAR x : real);
VAR c : char;
```

```
PROCEDURE a2 (y : char);
VAR d : integer;
```

```
BEGIN
```

```
...
```

```
END;
```

```
BEGIN
```

```
...
```

```
END;
```

```
PROCEDURE b1;  
VAR e : integer;  
BEGIN
```

```
...
```

```
END;
```

```
BEGIN
```

```
...
```

```
END.
```

- 1 Dentro del alcance del procedimiento *a2*, ¿a cuáles de las siguientes variables se puede hacer referencia?
  - a) solamente *d* y *y*.
  - b) solamente *d*, *y* y *c*.
  - c) solamente *a*, *b*, *c*, *d*, *x* y *y*.
  - d) ninguna de las anteriores.
- 2 Dentro del alcance del procedimiento *a1* ¿a cuáles de las siguientes variables se puede hacer referencia?
  - a) solamente *d*, *y* y *c*.
  - b) solamente *d*, *y*, *c*, *a*, y *b*
  - c) solamente *d*, *y*, *a*, *b*, *c* y *x*.
  - d) ninguna de las anteriores.
- 3 Dentro del alcance del procedimiento *b1*, ¿a cuáles de las siguientes variables se puede hacer referencia?
  - a) solamente *e*.
  - b) solamente *e*, *a* y *b*.
  - c) solamente *e*, *a*, *b* y *c*.
  - d) ninguna de las anteriores.
- 4 El programa principal podría llamar a los siguientes procedimientos:
  - a) solamente *a1* y *b1*.
  - b) solamente *a1*, *a2*, y *b1*.
  - c) solamente *a2* y *b1*.
  - d) ninguna de las anteriores.

- 5 Determinese la salida del siguiente programa si los datos de entrada incluyen solamente al entero 4

```

PROGRAMA principal (input, output);
VAR x : integer;
PROCEDURE a (VAR y : integer);
BEGIN
    y := y * 2
END;
PROCEDURE b (x : integer);
BEGIN
    x := x + 5
END;
BEGIN
    readln (x);
    b (x);
    writeln (x);
    a (x);
    writeln (x)
END.

```

Estúdiese el siguiente programa en Pascal para responder a las preguntas 6 a 9.

```

PROGRAM alcance (input, output);
VAR
    núm, reloj, temp: integer;
PROCEDURE pascal (núm: integer; VAR suma: integer);
VAR
    reloj : integer
BEGIN
    temp := 2 * núm + suma;
    reloj := temp * suma;
    núm := núm + 1;
    suma := suma + núm;
    writeln (temp, reloj, núm, suma)
END;
BEGIN
    núm := 1;
    reloj := 0;
    temp := 0;
    pascal (núm, temp);
    writeln (núm, reloj, temp);
    temp := temp + 1
    pascal (temp, núm);
    writeln (núm, reloj, temp);
END;

```

**6** Determíñese la salida del programa *alcance*.

**7** ¿Cuáles de los siguientes afirmaciones respecto al programa *alcalce* son ciertas?

- a) El alcance de la variable *temp* no incluye al procedimiento
- b) El programa principal no puede hacer referencia a la variable local *reloj* del procedimiento.
- c) El procedimiento se podría colocar después del programa principal.
- d) La variable local *reloj* es ilegal, ya que se declaró en el programa principal.
- e) El parámetro variable *suma* debe ir antes del parámetro de valor num.

**8** ¿Cuáles de las siguientes afirmaciones acerca del programa *alcance* son falsas?

- a) El procedimiento puede hacer referencia a la variable global *temp*.
- b) El nombre de procedimiento *pascal* no es un identificador válido.
- c) La variable local llamada *reloj* es un identificador válido.
- d) El alcance de la variable local *reloj* es el procedimiento.
- e) El *writeln* en el programa principal no exhibirá el contenido de la variable local *reloj* en el procedimiento.

**9** ¿Cuáles de las siguientes afirmaciones acerca del programa *alcance* son falsas?

- a) El parámetro de valor *núm* en la declaración del procedimiento es un parámetro formal.
- b) El parámetro *núm* en la proposición que llama al procedimiento es el parámetro verdadero.
- c) El parámetro formal *núm* es una variable local.
- d) La variable global *núm* es idéntica al parámetro formal *núm*.
- e) El alcance de la variable global *núm* no incluye al procedimiento.

**10** Determíñese la salida del siguiente programa.

```
PROGRAMA detectar (input, output);
VAR
    a, b, c, d : integer;
PROCEDURE pal (y : integer; VAR z : integer);
VAR
    c : integer;
BEGIN
    b := y * z;
    c := b + z;
    y := y + 1;
    z := z * (z + 1);
    writeln (b, c, y, z)
END;
BEGIN
    a := 2;
    c := 3;
    d := 5;
    pal (c, d);
```

```
writeln (a, b, c, d);
b := 4;
pal (b, a);
writeln (a, b, c, d)
END.
```

## SECCIÓN 7.2 FUNCIONES

Las *funciones* son subprogramas que calculan un solo valor. Por ejemplo, la función estándar *sqrt* calcula la raíz cuadrada de un número no negativo. En esta sección se estudiarán las funciones en Pascal, tanto las funciones estándar integradas, como las definidas por el usuario.

### Funciones estándar

Las funciones estándar son funciones predefinidas con que cuenta el Pascal estándar (véase el apéndice D). Ya se vieron las siguientes funciones aritméticas: *sqrt*, *sqr*, *abs*, *round* y *trunc* (véase el Cap. 2). Las funciones aritméticas que faltan son las funciones trigonométricas *sin*, *cos*, y *arctan* y las funciones exponencial y logaritmos natural *exp* y *ln*. Se volverá a estas funciones más adelante en esta sección.

Las funciones estándar incluyen también las cuatro funciones ordinales *pred*, *succ*, *ord* y *char*, las cuales se estudiarán en el siguiente capítulo.

Las demás funciones estándar son aquellas que producen resultados booleanos. Específicamente, estas funciones son:

- |            |   |
|------------|---|
| <i>odd</i> | Determina si su argumento es un entero <i>non</i>   |
| <i>eol</i> | Determina si el siguiente carácter en los datos de entrada es un carácter de fin de línea   |
| <i>eof</i> | Determina si el siguiente carácter en los datos de entrada es un carácter de fin de archivo |

Las funciones que producen resultados booleanos se llaman en ocasiones *predicados*. En esta sección se estudiarán con detalle todas estas funciones.

#### Función *odd*

La función *odd* produce el resultado *true* (verdadero) si el argumento entero es un número *non* y produce el resultado *false* si el argumento es un entero par. Por ejemplo, el siguiente segmento en Pascal espera que el usuario introduzca un entero y en seguida exhibe la palabra *NON* o *PAR* según sea apropiado. Se supone que *número* se declaró como variable entera.

```
read (número);
If odd (número);
THEN writeln ('NON')
ELSE writeln ('PAR')
```

```
read (número);
IF número MOD 2 <>0
THEN writeln ('NON')
ELSE writeln ('PAR')
```

La función *odd* es preferible, ya que hace más legible el programa. Obsérvese que también se puede utilizar la función *odd* para determinar si un número es par:

```
IF NOT odd (número)
THEN...
```

### Funciones *eoln* y *eof*

Los datos que se introducen cuando se está ejecutando el programa y los datos que se introducen en un archivo mediante un editor de textos deben terminar de tal manera que un programa que lea los datos pueda determinar cuándo ha llegado al final. Cuando se almacenan datos en disco o cinta, una centinela especial (agregado por el sistema de cómputo durante la creación original del archivo) marca el fin de los datos. Por otro lado, cuando se introducen los datos desde el teclado, lo normal es que el usuario tenga que agregar manualmente el centinela al apretar una tecla especial o un grupo de teclas (en muchos casos las teclas CONTROL y Z o CONTROL y D oprimidas simultáneamente). Cuando se emplea un programa en Pascal para crear un archivo de salida, el centinela se agrega automáticamente. Cuando un programa en Pascal encuentra un centinela de éstos en los datos de entrada, no se modifica variable alguna, sino que se toma nota del hecho de que se ha llegado al fin del archivo. Este centinela se llama *fin de archivo* (*end-of-file*), y la función estándar que detecta el hecho de haberlo encontrado se llama *eof*. En este texto el fin de archivo se representará así: <eof>.

Los archivos de texto no son meramente secuencias de caracteres de longitud arbitraria terminadas por el fin de archivo. Se subdividen en líneas, cada una de las cuales, cuando se introduce, normalmente se exhibe como un solo renglón de caracteres en la terminal. El número mínimo o máximo de caracteres que deben aparecer en una sola línea de texto no está definida previamente. (Es decir, el Pascal estándar no define un límite explícito, pero la mayor parte de los sistemas ponen un límite a la longitud máxima.) Por tanto, a todas las líneas se agrega un centinela especial conocido como carácter de *fin de línea*, el cual se representa en este texto mediante <eoln>. La función *eoln* permite al programador en Pascal determinar si el siguiente carácter en la entrada es el fin de línea. Puede pensarse que el carácter <eoln> se genera cuando se oprime la tecla RETURN o ENTER en la terminal (si se emplea un sistema interactivo).

Ambas funciones, *eof* y *eoln*, requieren un solo argumento, específicamente el nombre del archivo (la variable de archivo) que se está probando para detectar la condición especificada. El Pascal estándar permite omitir completamente el argu-

mento (y los paréntesis) si se desea probar el archivo estándar *input*. Por tanto, estas dos proposiciones normalmente son equivalentes:

```
IF eoln (input) THEN ...
IF eoln THEN ...
```

Los renglones que se presentan en seguida son la representación de un archivo de texto en donde se muestran explícitamente los fines de línea y de archivo. Obsérvese que cada línea contiene el nombre de una persona y un salario anual. Se hará referencia a este archivo cuando se analicen las funciones *eoln* y *eof*.

Juan Barrera	\$40000<eoln>
Sandra Frías	\$45000<eoln>
Ana María Escadillo	\$43713<eoln>
Edelia Aceves	\$51529<eoln>
<eof>	

Las funciones *eoln* y *eof* producen el resultado *true* cuando el siguiente carácter del archivo es <eoln> o <eof>. De manera similar, cuando el siguiente carácter del archivo *no* es <eoln> o <eof>, las funciones producen el resultado *false*. En seguida se verá cómo se puede usar la función *eoln* para leer y exhibir la primera línea del archivo de datos.

Cuando se invoca a la función *eoln* se examina el siguiente carácter del archivo (es decir, el que sigue al último carácter leído) sin leerlo realmente. Para usar la función *eoln* no se lee un carácter y se le examina después para determinar si es el carácter de fin de línea, sino que se pregunta si el siguiente carácter de los datos de entrada es el fin de línea.

He aquí entonces el código en Pascal que lee y exhibe la primera línea del archivo. Supóngase que *uncar* se declaró como variable de carácter.

```
WHILE NOT eoln DO (* mientras no se llegue al fin de línea *)
BEGIN
```

read (uncar);	(* capturar el siguiente carácter *)
write (uncar)	(* exhibirlo *)

```
END;
```

writeln;	(* "exhibir" un carácter de fin de línea *)
readln	(* y pasar por alto el fin de línea en la entrada *)

Es preciso hacer hincapié en un aspecto importante de este ejemplo. El ciclo WHILE por sí mismo no ilustra ningún concepto nuevo, excepto el uso de la función *eoln*. No obstante, cuando este ciclo termina, el carácter de fin de línea no se habrá leído. Se debe mencionar lo siguiente acerca del procesamiento de <eoln>:

- La función *eoln* prueba si existe el carácter <eoln>, pero no lo lee.
- La proposición *write* jamás puede escribir un carácter <eoln>.

En el código anterior, por tanto, se deben incluir las dos últimas proposiciones para escribir el <eoln> (el *writeln*) y para leer el <eoln> (el *readln*).

La razón de que no se pueda leer simplemente el <eoln> asignándolo a *uncar* para producir un fin de línea en la salida es que el <eoln> se transforma en un espacio en blanco (' ') cuando se lee. Esto garantiza que el programa funcione correctamente en todos los sistemas de Pascal estándar sin importar las técnicas posiblemente diferentes que se empleen para representar internamente el carácter <eoln>. Si se leyera y después exhibiera el carácter <eoln> sólo se encontraría un espacio adicional en la salida, pero *no un renglón nuevo*. Este hecho facilita el pasar por alto los caracteres <eoln> cuando no interesa la estructura de líneas del archivo de entrada. La única forma de detectar un carácter de fin de línea es la función *eoln*.

Ahora se tratará de leer y exhibir el archivo completo. Se utilizará la solución para una sola línea que ya se obtuvo y se agregará una prueba para detectar el fin de archivo mediante la función *eof*. Cuando se invoca la función *eof* se prueba el siguiente carácter del archivo de entrada para determinar si es el carácter de fin de archivo; nunca se le lee realmente. Cualquier intento de leer el carácter de fin de archivo producirá un mensaje diagnóstico (que suele ser fatal) y la terminación del programa sin más ni más. He aquí la solución al problema:

**WHITE NOT eof DO (\* mientras no se llegue al fin del archivo \*)**

**BEGIN**

**WHILE NOT eoln DO (\* mientras no se llegue al fin de línea \*)**

**BEGIN**

read (uncar);	(* capturar el siguiente carácter *)
write (uncar)	(* exhibirlo *)

**END;**

writeln;	(* "exhibir" un carácter de fin de línea *)
----------	---

readln	(* y pasar por alto el fin de línea en la entrada *)
--------	--

**END**

El ciclo principal en este segmento de programa dice, de hecho, que mientras no se llegue al final del archivo, se debe procesar una línea más. El ciclo interno es exactamente igual al problema anterior. Puesto que se supone que el archivo de entrada es el estándar en estos problemas, se permitió la omisión de los parámetros de las funciones; pero si se estuviera procesando un archivo diferente, sería preciso indicar explícitamente el nombre de la variable de archivo.

Una vez más, se recalca que el fin de archivo *no* se debe leer. Es por esta razón que se utilizó un ciclo WHILE en vez de un ciclo REPEAT-UNTIL. Es posible predecir el siguiente carácter en un archivo de entrada. (Es posible *anticipar* el si-

guiente carácter de un archivo de entrada, pero es necesario estar preparados para todas las eventualidades.)

Aunque no es común, en ocasiones es posible crear archivos sin un carácter de fin de línea antes de fin de archivo. Esto complica el procesamiento de un archivo de entrada, pero no lo imposibilita.

El uso de las funciones *eoln* y *eof* recién descrito es muy común en el procesamiento de caracteres de un archivo de entrada. En un capítulo posterior se volverá al tema de procesamiento de textos.

### Funciones definidas por el usuario

Las *funciones definidas por el usuario* en Pascal son similares a los procedimientos, con la excepción de que el objetivo de una función es devolver únicamente un valor al punto en que se le llamó. Por ejemplo, supóngase que se desea escribir una función llamada *cuarta* que produce la cuarta potencia de un número del tipo real. Es decir, si el número se llama *núm*, se desea calcular

$$\text{núm}^4 = \text{núm} \cdot \text{núm} \cdot \text{núm} \cdot \text{núm}$$

Por ejemplo,

$$1.0^4 = 1.0$$

$$2.0^4 = 16.0$$

$$3.5^4 = 150.0625$$

La siguiente función en Pascal realiza este cálculo:

**FUNCTION cuarta (número: real): real;**  
**(\* Calcular la cuarta potencia del argumento \*)**

**BEGIN**

cuarta := sqr (sqr (número))

**END;**

Obsérvese que el encabezado de la función se parece al encabezado de un procedimiento. La palabra reservada **FUNCTION** va seguida del identificador de la función y la lista de parámetros. Ésta va seguida de un signo de dos puntos ; el identificador de tipo que especifica el tipo de valor que va a tener el resultado de la función. En el caso de *cuarta*, el resultado será un valor real. El diagrama de sintaxis de las funciones en Pascal se muestra en la figura 7-4.

**Figura 7-4** Diagrama de sintaxis de las funciones.



El mecanismos que siguen las funciones en Pascal para especificar el valor que se devolverá es *asignar el valor al nombre de la función*. En la función *cuarta* la proposición de asignación

```
cuarta := sqr (sqr (número))
```

asigna la cuarta potencia de número al nombre de la función, con lo que se especifica que este valor es el que se debe devolver como valor de la función cuando ésta termine. Se pueden hacer varias asignaciones al nombre de la función, pero sólo se devolverá el último valor asignado. Tómese nota también de que *si el cálculo requiere la devolución de más de un valor, es preciso emplear un procedimiento en vez de una función*. Dado que las funciones proporcionan únicamente un resultado, y para hacer hincapié en el parecido entre las funciones matemáticas y las funciones de Pascal definidas por el usuario, por lo regular no se utilizan parámetros variables con las funciones.

Es importante no confundir el nombre de la función con una variable. Usar el nombre de la función en una expresión (p. ej., en el lado derecho de una proposición de asignación) no proporcionará el último valor asignado al nombre de función, sino más bien especificará que se debe invocar la función para producir un valor como resultado de la invocación.

En seguida se examinarán otros ejemplos. Supóngase que se requiere una función llamada *paridad* con dos parámetros de valor enteros llamados *núm1* y *núm2*, de manera que *paridad* proporciona el resultado *true* si *núm1* y *núm2* son ambos pares o ambos nones, y *false* en cualquier otro caso (es decir, si uno es non y el otro par). El código en Pascal que sigue es una solución del problema:

```
FUNCTION paridad (núm1, núm2: integer): Boolean;
(* Resulta true si núm1 y núm2 son ambos pares *)
(* o ambos nones, y false si no es así.           *)
BEGIN
  paridad := (odd (núm1) AND odd (núm2)) OR
             (NOT odd (núm1) AND NOT odd (núm2))
END;
```

Obsérvese el uso de la función *odd*. Una vez más, el tipo de dato del resultado debe seguir a la lista de parámetros y el signo de dos puntos. La función contiene una proposición de asignación con el nombre de la función en el lado izquierdo de la asignación. El valor resultante siempre se asigna al nombre de la función.

La siguiente función que se usará como ejemplo calcula la tangente de un ángulo. El Pascal estándar no incluye una función de tangente (aunque es posible que algunas versiones la ofrezcan como extensión), por lo que se usará la relación

$$\tan x = \frac{\sin x}{\cos x}$$

en la función. El código en sí es muy sencillo:

```
FUNCTION tan (x : real) : real;
BEGIN
    tan := sen (x) / cos (x)
END;
```

En una aplicación práctica conviene validar los argumentos de la función. Por ejemplo, algunos valores de  $x$  hacen que “tan ( $x$ )” produzca un error de ejecución (p. ej., cuando  $\cos (x) = 0$ ).

Por último, considérese una función para calcular el valor de  $a^b$ . Es decir, se requiere una función que calcule  $a$  elevado a la potencia  $b$ . Se usa la propiedad

$$a^b = e^{b \ln a}$$

(donde  $\ln a$  es el logaritmo natural de  $a$ ) para realizar la tarea de la función. He aquí el código en Pascal que usa las funciones estándar *exp* y *ln*:

```
FUNCTION potencia (a, b: real): real;
BEGIN
    potencia := exp (b * ln (a))
END;
```

### Invocación de funciones

Al igual que en el caso de los procedimientos, para invocar una función se requiere que el orden y tipo de datos de los parámetros verdaderos concuerden con el orden y tipo de datos de los parámetros formales. Si esto no se cumple, el compilador lo detectará e informará al usuario del error.

La invocación de funciones no es igual a la invocación de procedimientos. La razón de esta diferencia es que la función devuelve un valor al punto de invocación, aunque no a través de los parámetros mismos, como en el caso de los procedimientos. A resultas de esto, para invocar las funciones definidas por el usuario se escribe su nombre y lista de parámetros en una expresión, al igual que con las funciones estándar. Por ejemplo, para determinar el valor de dos a la tercera potencia y almacenar este valor en la variable  $x$ , se escribiría

`x := potencia (2.0, 3.0)`

y para calcular y exhibir la tangente de 17.3 se escribiría

`writeln ('Tangente de 17.3 = ', tan (17.3))`

sin olvidar que una proposición *write* o *writeln* puede exhibir el valor de una expresión.

Las invocaciones de funciones se tratan igual que cualquier otro valor del tipo que produce la función. Hasta es posible combinar funciones. Por ejemplo, para

```
y := potencia (tan (x), 3.0) / 2.0
```

A continuación se resolverá un problema de aplicación que utiliza una función definida por el usuario.

### Problema 7.2

*Escríbase un programa en Pascal que emplee una función para leer una lista de  $N$  calificaciones de examen (enteros no negativos) y encuentre la calificación más alta. Si la calificación más alta es mayor de 90, exhibase un mensaje que indique este hecho. El valor de  $N$  será el primero de los datos.*

Ahora aparece un programa completo en Pascal llamado *máximo*. Este programa utiliza una función de nombre *alta*. El número de valores  $N$  se leerá en el programa principal y se pasará como parámetro a la función. En la función *alta* se leen las  $N$  calificaciones y se usan, pero no se proporciona el programa que la llamó. Esto es aceptable, ya que las calificaciones no se necesitan fuera de la función.

PROGRAM *máximo* (input, output);

(\* Determinar la más alta de  $N$  calificaciones capturadas \*)  
(\* y exhibir un mensaje si es mayor de 90. \*)

VAR

número: integer;  
FUNCTION alta (N: integer): integer;  
(\* Leer y devolver el mayor de  $N$  enteros \*)

VAR

índice: integer;  
calif: integer;  
máx: integer;

BEGIN

máx := 0; (\* dar valor inicial a la calificación más alta \*)

FOR índice := 1 TO N DO

BEGIN

write ('teclee una calificación:');

readln (calif);

IF calif > máx

THEN máx := calif (\* nueva calificación más alta \*)

END;

```

alta := máx (* devolver calificación más alta *)
END;
(* Programa principal *)
BEGIN
  write ('Especifique el número de calificaciones de examen: ');
  readln (número);
  IF alta (número)>90 (* invocación de la función *)
    THEN writeln ('La calificación más alta es superior a 90.')
END.

```

Recuérdese que el nombre de la función debe aparecer en el lado izquierdo de por lo menos una proposición de asignación dentro de la función misma para comunicar el resultado de la función al punto de invocación. Por lo general, el nombre de la función no deberá aparecer en el lado derecho de una proposición de asignación dentro de la función misma, como se indicó anteriormente. Por ejemplo, supóngase que se quisiera escribir una función para calcular la décima potencia de un entero. Es decir, si  $x$  es un argumento entero, el resultado que se desea es  $x^{10}$ . Supóngase que se llama a esta función *potencia10*. La siguiente función en Pascal *no* resolverá el problema, puesto que el nombre de la función aparece en el lado derecho de la proposición de asignación.

```

FUNCTION potencia10 (x: integer): integer;
(* Función que NO funciona *)
VAR
  i: integer;
BEGIN
  potencia10 := 1; (* asignar valor inicial al resultado *)
  FOR i:= 1 TO 10 DO
    potencia10 := potencia10 * x
END;

```

Para modificar la función se puede agregar una variable temporal con el fin de guardar el resultado parcial agregando después una proposición para asignar el resultado final al nombre de la función. He aquí la solución modificada y correcta:

```

FUNCTION potencia10 (x: integer): integer;
(* Devolver x a la décima potencia *)
VAR
  i, temp: integer;
BEGIN
  temp := 1; (* asignar valor inicial al resultado *)
  FOR i:= 1 TO 10 DO
    temp := temp * x;
  potencia10 := temp
END;

```

Cuando aparece el nombre de la función en una expresión, se invoca la función. Esto se cumple aun en el caso de expresiones dentro de la función. Las funciones que se invocan a sí mismas se llaman funciones *recursivas*. En la siguiente sección se examinarán algunos ejemplos.

## EJERCICIOS DE LA SECCIÓN 7.2

- 1 Determinese el resultado de estas invocaciones de función
  - a) odd (2)
  - b) odd (0)
  - c) odd (-3)
  - d) odd (5-1)
  - e) odd (2 \* trunc (3.6))
- 2 Examínense el siguiente programa y los datos de entrada. Determinese qué es lo que se exhibirá cuando se ejecute el programa.

**PROGRAM prueba (input, output);**

**VAR**

    número: integer;

**BEGIN**

```
        read (número);
        WHILE NOT eoln DO
            BEGIN
                writeln (número);
                read (número)
            END
        END.
```

Datos de entrada:

```
2    4    5<eoln>
6<eoln>
<eof>
```

- 3 Examínese el siguiente programa y los datos de entrada. Determinese qué es lo que se exhibirá cuando se ejecute el programa.

**PROGRAM probar (input, output);**

**VAR**

    núm1, núm2, cuenta: integer;

**BEGIN**

```
        cuenta := 0;
        WHILE NOT eof DO
            BEGIN
                readln (núm1, núm2);
                cuenta := cuenta + 1
            END;
```

```
writeln ('La cuenta es', cuenta: 1)
END.
```

Datos de entrada:

```
2    4    5<eoln>
6    7<eoln>
5    10   6<eoln<
<eof>
```

- 4 ¿Cuáles de las siguientes funciones proporcionarían de manera correcta la porción entera de  $x$ , donde  $x$  se define como variable real?

a) FUNCTION prueba (VAR x: real);

```
BEGIN
  x := trunc(x)
END;
```

b) FUNCTION prueba (x): integer;

```
BEGIN
  prueba := trunc(x)
END;
```

c) FUNCTION prueba (x: real): integer;

```
BEGIN
  prueba := trunc(x)
END;
```

d) FUNCTION prueba (x: real): integer;

```
BEGIN
  x := trunc(x)
END;
```

e) FUNCTION prueba (x: real): integer;

```
BEGIN
  trunc(x)
END;
```

- 5 Examíñese el siguiente segmento de programa:

```
PROCEDURE trueque (VAR x, y: integer);
VAR temp: integer;
BEGIN
  temp := x;
  x := y;
  y := temp
END;
```

```

FUNCTION mult (x: integer): integer;
BEGIN
    mult := x * x
END;

```

¿Cuáles de las siguientes son invocaciones correctas del procedimiento y/o función anteriores si  $x$ ,  $y$  y  $z$  son variables enteras?

- a) trueque (mult(x), x);
- b) trueque (3, 4);
- c) mult (10);
- d) y := mult (10);
- e) z := trueque (x, y);

6 Determinese qué es lo que se exhibe cuando se ejecuta el siguiente programa.

```

PROGRAM final (input, output);
VAR
    manzanas, plátanos, naranjas: integer;
PROCEDURE grado (a, b: integer; VAR c: integer);
VAR
    manzanas: integer;
FUNCTION que (d: integer): integer;
BEGIN
    que := d * 2
END;
BEGIN
    b := que (a);
    manzanas := 16;
    c := que (a)
END;
BEGIN
    manzanas := 2
    platános := 3
    naranjas := 6
    grado (manzanas, plátanos, naranjas);
    writeln (manzanas, plátanos, naranjas)
END.

```

7 Escríbase una función en Pascal llamada *digfinal* que espere un solo parámetro entero positivo. *Digfinal* da como resultado el último dígito de su parámetro. Por ejemplo, “Digfinal (2456)” debe dar como resultado 6.

- 8 Escribase un programa en Pascal que lea un número desconocido de calificaciones (entre cero y 100, inclusive) y exhiba el número total de calificaciones y la calificación más baja. Supóngase que en cada línea hay una calificación y que los datos terminan con el fin de archivo.

## SECCIÓN 7.3 PROCEDIMIENTOS Y FUNCIONES RECURSIVOS

Una función o procedimiento que se llama o invoca a sí mismo se dice que es *recursivo*. El lector podría pensar que las funciones recursivas son curiosidades y que no son muy útiles. Sin embargo, las funciones recursivas (y también los procedimientos recursivos) pueden ser muy eficientes para la resolución de problemas, sobre todo en aplicaciones de computación avanzada. En esta sección se examinarán algunos ejemplos y aplicaciones elementales para dar al lector una idea de la elegancia y sencillez de las funciones recursivas.

Las funciones recursivas son apropiadas cuando se puede definir un problema en términos de sí mismo (de manera recursiva). Por ejemplo, considérese la siguiente definición de un problema:

*Una institución financiera paga un interés anual del 12% al principio de cada año sobre el capital que se ha dejado en depósito durante el año anterior. Se desea saber cuánto valdría una inversión inicial de \$1000 si ella y el interés se dejaran durante n años.*

Este problema se puede volver a escribir en forma recursiva si se hacen las siguientes observaciones. Sea  $a_n$  la cantidad de dinero acumulada al principio del año  $n$ . Entonces

$$\begin{aligned} a_0 &= 1000 && \text{(inversión inicial)} \\ a_1 &= 1000 + 0.12 * 1000 && \text{(capital al final del primer año)} \\ a_2 &= a_1 + 0.12 * a_1 && \text{(capital al final del segundo año)} \end{aligned}$$

En general, puede verse que

$$a_i = a_{i-1} + 0.12 * a_{i-1}$$

es la cantidad invertida al final del año  $i$ . Si se simplifica esta expresión al sumar los términos se tiene

$$a_i = 1.12 * a_{i-1}$$

y el enunciado del problema produce la siguiente definición recursiva

$$\begin{aligned} a_0 &= 1000 \\ a_i &= 1.12 * a_{i-1} \end{aligned}$$

Ahora el problema es cómo calcular  $a_n$ , dado el valor de  $n$ . Por ejemplo, si  $n = 3$ , ¿cuánto dinero se habrá acumulado al final del tercer año? Si se sustituye  $n$  en la definición recursiva, se sabe que

$$a_3 = 1.12 * a_2$$

pero también se sabe que

$$a_2 = 1.12 * a_1$$

y que

$$a_1 = 1.12 * a_0$$

Pero, como ya se conoce el valor de  $a_0$ , es posible calcular  $a_1$ ,  $a_2$  y  $a_3$ .

$$a_1 = 1.12 * 1000 = 1120$$

$$a_2 = 1.12 * 1120 = 1254.40$$

$$a_3 = 1.12 * 1254.40 = 1404.93$$

El Pascal puede hacer todos estos cálculos y para ello basta con transformar el enunciado recursivo del problema en una función recursiva en Pascal. Recuérdese la definición recursiva del problema:

$$a_0 = 1000, \quad \text{y} \quad a_i = 1.12 * a_{i-1}$$

Si se desea calcular el valor de  $a_n$  para un valor arbitrario de  $n$  (posiblemente cero), se escribirá el siguiente pseudocódigo:

Si  $n = 0$

Entonces  $a_n = 1000$

En caso contrario  $a_n = 1.12 * a_{n-1}$

Esto se puede convertir directamente en una función en Pascal. Basta con agregar la definición usual de parámetros (para  $n$ ) y sustituir el nombre  $a_n$  por una variable del Pascal. He aquí entonces la función en Pascal completa para determinar el valor de la inversión después de  $n$  años:

```
FUNCTION capital (n: integer): real;
(* Función recursiva para calcular el capital en una cuenta que *)
(* paga interés del 12% con una inversión inicial de $1000.00. *)
BEGIN
  If n = 0
  THEN capital := 1000.0
  ELSE capital := 1.12 * capital (n-1)
END;
```

Conviene hacer dos observaciones importantes en este punto. Tómese nota del nombre de la función en el lado derecho con el parámetro verdadero. Esto hará que la función *capital* se llame a sí misma. Obsérvese además la forma como la solución en Pascal refleja la definición recursiva del problema. Esta correspondencia tan cercana entre la función recursiva en Pascal y el enunciado del problema permite usar fácilmente Pascal para resolver problemas recursivos. De hecho, el problema de inversión que se acaba de examinar se puede escribir en forma más concisa sin recursión (véase el Prob. 3 al final del capítulo). Sin embargo, la solución recursiva es más adecuada porque refleja la solución al problema de manera más natural.

¿Cómo es que la computadora calcula realmente el resultado en la función recursiva *capital*? Estúdiese el cálculo a mano que se hizo antes para  $n = 3$ . Esa ilustración deberá dar al lector una idea de la técnica que emplea la computadora. Cuando encuentra una llamada de función recursiva, debe postergar temporalmente el cálculo para evaluar la función recursiva. La computadora guarda toda la información que necesita para continuar el cálculo después de la llamada recursiva, y una vez que se obtiene el resultado de la llamada recursiva, continúa el cálculo de la función. Este proceso se repite aunque existan varios niveles de llamadas recursivas. Imagínese el número de niveles requerido para determinar el saldo de la cuenta después de 50 años. La computadora mantendrá con exactitud el conjunto completo de resultados temporales hasta obtener el resultado final.

Cuando se invoca un procedimiento o función, sea en forma recursiva o no, los valores de las variables y constantes locales del segmento de programa que llama se guardan en una *pila*. Las pilas se estudiarán en el capítulo 13, pero por el momento el lector puede considerarlas como parecidas al mecanismo que se utiliza en ocasiones para entregar platos en una cafetería. El plato disponible es el que está arriba y, cuando se saca, todos los demás son empujados hacia arriba. Cuando se agrega un plato a la pila todos los platos presentes en ese momento se empujan hacia abajo. Lo mismo sucede con las invocaciones de funciones y procedimientos. Al invocarse un procedimiento, los valores locales del segmento que llama se guardan en una pila y cuando termina (es decir, cuando devuelve el control al punto en que se invocó el procedimiento), los valores locales se recuperan de la pila. Este procedimiento se usa para todas las funciones y procedimientos, sean o no recursivos.

Ahora se examinará un problema diferente que se puede resolver mediante el uso de funciones recursiva.

### Problema 7.3

*Escribase una función recursiva para calcular la suma de los enteros 1, 2, 3, ..., n. Es decir, calcúlese recursivamente  $1 + 2 + 3 + \dots + n$*

En primer lugar se definirá recursivamente el problema. Si se llama  $S_n$  a la suma de los  $n$  primeros enteros, se tiene que

$$S_0 = 0, \quad \text{y} \quad S_i = i + S_{i-1}$$

$$\begin{aligned}
 S_5 &= 5 + S_4 \\
 &= 5 + 4 + S_3 \\
 &= 5 + 4 + 3 + S_2 \\
 &= 5 + 4 + 3 + 2 + S_1 \\
 &= 5 + 4 + 3 + 2 + 1 + S_0 \\
 &= 5 + 4 + 3 + 2 + 1 + 0 \\
 &= 15
 \end{aligned}$$

La solución en Pascal refleja una vez más la definición recursiva del problema:

```

FUNCTION suma (n: integer): integer;
(* Función recursiva para calcular la suma *)
(* de los enteros 1, 2, 3, ..., n *)
BEGIN
  IF n = 0
  THEN suma := 0
  ELSE suma := n + suma (n-1)
END;

```

La solución del siguiente problema ilustrará el uso de funciones anidadas con una función recursiva. Este problema ya se resolvió dos veces, la primera vez usando selección (Sec. 5.4) y la segunda con ciclos (Sec. 6.2).

#### Problema 7.4

*Los datos de entrada contienen el número de mes y día del mes en un año no bisiesto. Determinese el día del año (en la escala de 1 a 365) mediante funciones de Pascal.*

Es menester calcular el número total de días en los meses que preceden al mes estipulado y sumar a este total el día del mes. La función principal *dianual* incluirá una sola proposición que logra lo anterior. Para calcular el número total de días en los meses previos al mes especificado se invocará una función recursiva llamada *díasenmes*. Esta función invocará a su vez otra función llamada *díamensual* que da como resultado el número de días en un mes dado. La definición recursiva para calcular *díasenmes* es la siguiente (donde  $m_n$  denota el número total de días en los meses previos al mes  $n$ ):

$$\begin{aligned}
 m_1 &= 0 \text{ (no hay días previos a enero)} \\
 m_n &= m_{n-1} + \text{días en el mes número } (n-1)
 \end{aligned}$$

Ahora es posible escribir las funciones. La siguiente función en Pascal resolverá el problema mediante una función recursiva.

```

FUNCTION dianual (mes, día: integer): integer;
(* Función para calcular el día del año, dados el mes y el día *)
(* El mes debe estar en la escala de 1 a 12 *)
FUNCTION diámensual (número: integer): integer;
(* Produce el número de días en el mes especificado *)
BEGIN
  CASE número OF
    1, 3, 5, 7, 8, 10, 12 : diámensual := 31;
    2                      : diámensual := 28;
    4, 6, 9, 11            : diámensual := 30
  END
END;

FUNCTION díasenmes (mes: integer): integer;
(* Función recursiva que produce el número total de *)
(* días en los meses anteriores al mes del argumento *)
BEGIN
  IF mes = 1
  THEN díasenmes := 0
  ELSE díasenmes := díasenmes (mes - 1) + diámensual (mes - 1)
END

(* Función principal *)
BEGIN
  dianual := díasenmes (mes) + día
END;

```

### Recursión e iteración

Los ejemplos presentados en esta sección se pueden escribir sin emplear recursión. Por ejemplo, la función *capital* se puede escribir en forma no recursiva, o *iterativa*, como sigue;

```

FUNCTION capital (n: integer): real;
(* Función no recursiva o iterativa para calcular *)
(* el saldo en una cuenta al 12% de interés anual *)
(* con una inversión inicial de $1000.00 *)
VAR
  temp : real;
  índice: integer;
BEGIN
  temp := 1000.00;
  FOR índice := 1 TO n DO
    temp := temp * 1.12;
    capital := temp
END;

```

Este método de calcular se llama *iteración* y no es recursivo. Si un lenguaje de programación como el Pascal permite la recursión y la iteración, ¿cuál de los dos métodos se debe usar para resolver un problema dado? Recuérdese que en una solución recursiva la computadora debe llevar la pista de todas las llamadas recursivas, y esto puede requerir una gran cantidad de tiempo y de memoria. Por esto, las soluciones recursivas a los problemas muchas veces son menos eficientes que las soluciones iterativas. En general, se debe usar una función recursiva únicamente si la solución no se puede traducir fácilmente a la solución iterativa equivalente o si la eficiencia de la solución recursiva es satisfactoria.

## EJERCICIOS DE LA SECCIÓN 7.3

- 1 Determíñese qué es lo que se exhibe cuando se ejecuta el siguiente programa.

```
PROGRAM recursivo1 (input, output);
FUNCTION incógnita (núm : integer):integer;
BEGIN
  IF núm = 0
  THEN incógnita := 5
  ELSE incógnita := incógnita (núm - 1) + 1
END;
BEGIN
  writeln (incógnita (10))
END.
```

- 2 Determíñese qué es lo que se exhibe cuando se ejecuta el siguiente programa.

```
PROGRAM recursivo2 (input, output);
FUNCTION que (núm: integer): integer;
BEGIN
  IF núm = 1
  THEN que := 10
  ELSE que := que (núm-1) + 10
END;
BEGIN
  writeln (que (5))
END.
```

- 3 La función  $f(n) = n!$  calcula el producto de los enteros 1, 2, 3, ...,  $n$ . Es decir,

$$n! = 1 \cdot 2 \cdot 3 \cdots n$$

Por ejemplo,  $3! = 1 \cdot 2 \cdot 3 = 6$ .

- a) Defínase recursivamente a  $n!$   
 b) Escríbase una función recursiva en Pascal para calcular  $f(n) = n!$
- 4 La función *potencia* ( $x, n$ ) =  $x^n$ , donde  $n > 0$ , calcula  $x \cdot x \cdot x \cdots x$  ( $n$  factores de  $x$ ), para un número real  $x$ . Por ejemplo, *potencia* (4.0, 3) = 64.0.
- a) Escríbase una función no recursiva en Pascal para calcular *potencia* ( $x, n$ ).  
 b) Defínase  $x^n$  en forma recursiva.  
 c) Escríbase una función recursiva en Pascal con dos parámetros  $x$  y  $n$  para calcular  $x^n$ .
- 5 La secuencia de Fibonacci se define de manera recursiva como sigue:

$$\begin{aligned}f_1 &= 1 \\f_2 &= 1 \\f_n &= f_{n-1} + f_{n-2}\end{aligned}$$

Los primeros 10 términos de la secuencia son:

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

Escríbase una función recursiva en Pascal para calcular el *enésimo* número de Fibonacci  $f_n$ .

## SECCIÓN 7.4 RESOLUCIÓN DE PROBLEMAS MEDIANTE PROCEDIMIENTOS Y FUNCIONES

En esta sección se aplicarán algunas de las técnicas vistas en este capítulo a un problema específico, la creación de un juego sencillo para enseñar la multiplicación.

### Problema de juego de multiplicación

Un maestro de escuela primaria quisiera escribir un programa de juego que ayude a los niños de su grupo a memorizar la tabla de multiplicación que se muestra en seguida.

x	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81



Figura 7-5 Diseño descendente del juego de multiplicación.

La parte medular de este problema es generar pares de enteros en forma repetida, ambos en la escala de cero a nueve. Estos números se exhibirán ante el niño para que los lea y los multiplique. Si el niño responde correctamente 10 ejercicios seguidos, “ganará” el juego. En caso contrario no ganará. En todos los casos se da al niño la opción de jugar otra vez.

Este problema se puede subdividir en tres subproblemas (véase la Fig. 7-5):

SUBPROBLEMA 1: Generar en forma aleatoria los ejercicios de multiplicación.

SUBPROBLEMA 2: Exhibir el problema, obtener la respuesta del niño y verificar si es correcta.

SUBPROBLEMA 3: Exhibir los resultados y procesar la opción de continuación.

Se utilizarán los siguientes procedimientos para resolver los problemas:

- 1 El procedimiento *ejercicio* genera en forma aleatoria un ejercicio de multiplicación (sin parámetros de valor y con dos parámetros VAR, cada uno de los cuales recibe un entero al azar en la escala de cero a nueve).
- 2 El procedimiento *prueba* valida la respuesta del niño y actualiza la calificación de acuerdo con el resultado (los parámetros de valor son los dos enteros al azar del procedimiento *ejercicio* y la respuesta del niño; los parámetros VAR son la calificación actualizada y una indicación de correcto/incorrecto).
- 3 El procedimiento *exhibir* exhibe los resultados y da al niño la opción de jugar otro juego (el parámetro de valor es la calificación; el parámetro VAR es la indicación de continuación).

El algoritmo emplea dos ciclos, el ciclo externo para continuar jugando y el ciclo interno para continuar generando ejercicios. Se puede escribir el algoritmo así:

*Algoritmo del juego de multiplicación*

Mientras se vayan a jugar más juegos

Mientras respuesta correcta y número de aciertos < 10:

    Generar un ejercicio de multiplicación.

    Validar la respuesta.

    Actualizar la calificación.

    Indicar situación de juego ganado/perdido y dar opción de otro juego.

Después de estudiar el algoritmo, se ve que el problema más difícil es generar en forma aleatoria los ejercicios de multiplicación. Se hará una disgresión breve para considerar con detalle este problema.

## Generadores de números aleatorios

Las programas o funciones que generan números aleatorios son muy comunes en los sistemas de cómputo y a menudo están incluidos en la biblioteca del sistema. Muchos sistemas de Pascal tienen funciones predefinidas para generar números aleatorios. Puesto que este libro hace hincapié en la transportabilidad y el Pascal estándar, se ha incluido una función generadora de números aleatorios que deberá funcionar en la mayor parte de los sistemas de Pascal.

Los generadores de números aleatorios producen normalmente una secuencia de números reales (casi siempre entre cero y uno) que para la mayor parte de las aplicaciones prácticas se pueden considerar como aleatorios. En realidad, puesto que los números se generan por programa, no son verdaderamente aleatorios. De hecho, los números generados se llaman muchas veces *números seudoaleatorios*.

Casi todos los generadores de números aleatorios parten de un valor inicial llamado *semilla*. El resto de los números aleatorios se generan una vez especificada la semilla.

Los generadores de números aleatorios pueden comenzar con la misma semilla cada vez que se ejecuta el programa. Esto es útil para depurar el programa, ya que es posible volver a generar los números aleatorios. No obstante, es posible cambiar la semilla cada vez que se ejecuta el programa, ya sea introduciendo una semilla diferente como dato de entrada o mediante una función no estándar (como el reloj del sistema).

He aquí una función generadora de números aleatorios escrita en Pascal estándar que funcionará en la mayor parte de los sistemas de cómputo, incluso en microcomputadoras. El lector no debe preocuparse demasiado en este punto si no comprende su funcionamiento. Existen muchos textos excelentes que analizan con detalle las funciones generadoras de números seudoaleatorios (p. ej., véase Knut, *The Art of Computer Programming* (Vol. 2): *Seminumerical Algorithms*).

```

FUNCTION azar (VAR semilla : real) : real;
(* La semilla es cualquier número real positivo menor que m, *)
(* constante cuyo valor se indica en seguida. *)  

CONST
  a : 93.0;
  m : = 8192.0;
  c = 1.0;
BEGIN
  (* Obsérvese que la modificación de semilla es efecto secundario *)
  semilla := a * semilla + c;
  semilla := round (((semilla/m) - trunc (semilla/m)) * m);
  azar := semilla/m
END;
```

Para emplear esta función es preciso asignar un valor inicial a la semilla en el programa principal e invocar la función. Téngase en mente que la semilla se emplea tanto en el programa principal como en la función y, por tanto, debe ser parámetro variable. He aquí un ejemplo de programa que genera 10 números aleatorios con este generador de números aleatorios y partiendo de una semilla de 0.7823.

---

```

PROGRAM pruebazar (output);
(* Programa para probar el generador de números aleatorios *)
VAR
    índice : integer;
    semilla : real;

FUNCTION azar (VAR semilla : real) : real;
(* La semilla es cualquier número real positivo menor que m, *)
(* constante cuyo valor se indica en seguida. *)
CONST
    a : 93.0;
    m := 8192.0;
    c = 1.0;

BEGIN
    (* Obsérvese que la modificación de semilla es efecto secundario *)
    semilla := a * semilla + c;
    semilla := round (((semilla/m) - trunc (semilla/m)) * m);
    azar := semilla/m

END;
(* programa principal *)
BEGIN
    semilla := 0.7823;
    FOR índice := 1 TO 10 DO
        writeln ('Un número aleatorio es', azar (semilla))

END.

```

---

Ahora se muestra la salida que produce este programa en un sistema de cómputo determinado. Si el usuario lo ejecuta en su sistema, es probable que obtenga resultados diferentes.

```

Un número aleatorio es 9.03320E-03
Un número aleatorio es 8.40210E-01
Un número aleatorio es 1.39648E-01
Un número aleatorio es 9.87427E-01

```

Un número aleatorio es 8.30811E-01  
 Un número aleatorio es 2.65503E-01  
 Un número aleatorio es 6.91895E-01  
 Un número aleatorio es 3.46313E-01  
 Un número aleatorio es 2.07275E-01  
 Un número aleatorio es 2.76733E-01

Puede verse que los números seudoaleatorios que genera este programa son números reales mayores o iguales que cero y menores que uno. En el programa que se va a resolver es preciso generar enteros entre cero y nueve. Para transformar los números reales en enteros se pueden multiplicar por 10 desechando después la parte fraccionaria. La multiplicación por 10 da lugar a un número real mayor o igual que cero y menor que 10. Es decir, el número tiene la forma  $x.yyyyy$ , donde  $x$  es 0, 1, 2, ..., 9 y  $yyyyy$  es una parte fraccionaria arbitraria. Por tanto, la expresión que se emplea para generar un entero al azar de un solo dígito es

`trunc(10 * azar (semilla))`

Si se aplica esto a los 10 números al azar generados anteriormente, se obtienen los siguientes 10 enteros:

0 8 1 9 8 2 6 3 2 2

Ahora volverá a la resolución en computadora del problema de juego de multiplicación.

### Solución completa al problema de juego de multiplicación

A continuación se muestra el programa completo en Pascal que resuelve este problema, junto con la salida que resulta de una ejecución ejemplo. Tómese nota del uso de variables booleanas para terminar la ejecución de los ciclos. Además, si el sistema del usuario cuenta con un generador de números aleatorios, puede sustituirse la función `azar` por una declaración de función del generador de números aleatorios del sistema propio.

**PROGRAM** juego (**input, output**);  
 (\* Juego de multiplicación \*)

**VAR**

número1, número2,	(* números a multiplicar *)
calif,	(* número de respuestas correctas *)
respuesta : integer;	(* respuesta del niño *)
bien,	(* ¿es respuesta = número1 * número2? *)
más juegos : boolean;	(* ¿jugar otra vez? *)
semilla : real;	(* para el número aleatorio *)

FUNCTION azar (VAR semilla : real) : real;  
(\* La semilla es cualquier número real positivo menor que m, \*)  
(\* constante cuyo valor se indica en seguida. \*)

303

PROCEDIMIENTOS  
Y FUNCIONES

CONST

a : 93.0;  
m := 8192.0;  
c = 1.0;

BEGIN

(\* Obsérvese que la modificación de semilla es efecto secundario \*)  
semilla := a \* semilla + c;  
semilla := round (((semilla/m) - trunc(semilla/m)) \* m);  
azar := semilla/m

END;

PROCEDURE ejercicio (VAR num1, num2: integer; VAR semilla: real);

(\* Generar el siguiente ejercicio de multiplicación para el niño \*)

BEGIN

num1 := trunc (10 \* azar (semilla));  
num2 := trunc (10 \* azar (semilla));

END;

PROCEDURE prueba (num1, num2, respuesta; integer;

VAR correcto : boolean;

VAR calif : integer);

(\* Validar la respuesta del niño y actualizar la calificación \*)

VAR

producto: integer; (\* num1 \* num2 \*)

BEGIN

producto := num1 \* num2; (\* generar respuesta correcta \*)  
IF respuesta = producto (\* ¿fue correcta la respuesta? \*)

THEN BEGIN

calif := calif + 1; (\* si, actualizar calificación \*)

correcto := true;

writeln ('Correcto') (\* y decírselo al niño \*)

END

ELSE BEGIN

correcto := false; (\* equivocación \*)

```

        write ('Lo siento, te equivocaste.');
        write ('La respuesta correcta es ');
        writeln (producto:1, '.')
        END

END;

PROCEDURE exhibir (calif: integer; VAR másjuegos: Boolean);
(* Exhibir resultados del juego y determinar nueva situación del mismo *)

VAR

    car: char;      (* Respuesta a la pregunta de juego nuevo *)

BEGIN

    IF calif = 10
    THEN writeln ('Tu calificación es un 10 perfecto, ¡magnífico!')
    ELSE writeln ('Tu calificación es ', calif:1,'.Prueba otra vez.');
    car := ' ';
    WHILE (car<>'S') AND (car<>'S') AND (car<>'N')
          AND (car<>'n') DO

BEGIN

    Write ('¿Otro juego (responde S o N)? ');
    readln (car)

END;

másjuegos := (car = 'S') OR (car = 's')

END;

(* Programa principal del juego de multiplicación *)

BEGIN

    semilla := 0.7823;      (* dar valor inicial a las variables *)
    más juegos := true;
    WHILE másjuegos DO

BEGIN

    writeln ('Se inicia un juego nuevo.');
    calif := 0;
    bien := true;
    WHILE (calif<10) AND bien DO

```

```
ejercicio (número1, número2, semilla);
write ('¿Cuándo es', número1:1, 'por', número2:1, '?');
readln (respuesta);
prueba (número1, número2, respuesta, bien, calif)

END;
exhibir (calif, másjuegos)

END.
```

---

*Programa juego*

---

Se inicia un juego nuevo.

¿Cuánto es 0 por 8? **0**  
Correcto  
¿Cuánto es 1 por 9? **9**  
Correcto  
¿Cuánto es 8 por 2? **16**  
Correcto  
¿Cuánto es 6 por 3? **36**  
Lo siento, te equivocaste. La respuesta correcta es 18.  
Tu calificación es 3. Prueba otra vez.  
¿Otro juego (responde S o N)? **S**  
Se inicia un juego nuevo.  
¿Cuánto es 2 por 2? **4**  
Correcto  
¿Cuánto es 7 por 4? **28**  
Correcto  
¿Cuánto es 5 por 7? **37**  
Lo siento, te equivocaste. La respuesta correcta es 35.  
Tu calificación es 2. Prueba otra vez.  
¿Otro juego (responde S o N)? **K**  
¿Otro juego (responde S o N)? **N**

---

*Programa juego: ejemplo de ejecución*

## SECCIÓN 7.5 TÉCNICAS DE PRUEBA Y DEPURACIÓN

En esta sección se analizarán algunos de los problemas que se pueden presentar cuando se usan procedimientos y funciones.

**Efectos secundarios**

Las funciones deben realizar alguna tarea usando los parámetros de entrada y produciendo un solo valor. Sin embargo, dado que las funciones son similares a los procedimientos, pueden hacer lo mismo que ellos. Por ejemplo, las funciones pueden tener parámetros variables además de los parámetros de valor en la lista de parámetros formales. Las funciones también pueden modificar el contenido de una variable global y ejecutar proposiciones de entrada/salida. Cuando se emplean estas operaciones en funciones se les denomina *efectos secundarios*. En la mayor parte de los casos conviene evitar los efectos secundarios (una excepción fue la función generadora de números aleatorios de la Sec. 7.4). Los efectos secundarios pueden provocar errores de programación que resultan difíciles de encontrar. Toda la información que se intercambie con un procedimiento o una función debe incluirse en la lista de parámetros, no en las variables globales. Esto permitirá que el procedimiento o función sea un módulo autosuficiente e independiente, susceptible de probarse y depurarse por sí solo. No hay que preocuparse de que otras partes del programa afecten al procedimiento o función de alguna manera sutil.

**Declaración adelantada**

Recuérdese que los procedimientos no se pueden invocar si la declaración del procedimiento no aparece antes de la proposición que lo invoca. No obstante, existen casos en los que puede ser necesario llamar a un procedimiento que no se ha declarado antes de la proposición que lo invoca. Por ejemplo, se pueden tener los procedimientos *uno* y *dos*, cada uno de los cuales llama al otro.

```
PROCEDURE uno (a: integer; VAR b: real);
declaraciones...
BEGIN
```

```
proposiciones...
dos (parámetros verdaderos);
proposiciones...
```

```
END;
PROCEDURE dos (x : integer; VAR y : real);
declaraciones...
BEGIN
```

```
proposiciones...
uno (parámetros verdaderos);
proposiciones...
```

```
END;
```

Como se indicó anteriormente, el procedimiento *dos* es correcto porque hace referencia al procedimiento *uno*, que ya se definió. Pero el procedimiento *uno* es in-

correcto, ya que hace referencia al procedimiento *dos*, que no se ha definido todavía. Invertir el orden de las declaraciones no resuelve el problema; en ese caso el procedimiento *dos* heredará el error.

El Pascal permite cambiar el orden usual de las declaraciones mediante una *declaración adelantada*. En esencia, se agrega la palabra *forward* (adelante) al encabezado completo del procedimiento. Esta declaración adelantada se coloca antes del procedimiento que llama, y la declaración real del procedimiento llamado (declaraciones locales y proposiciones) no contiene la lista de parámetros. Los procedimientos *uno* y *dos* se pueden escribir si se coloca la declaración adelantada del procedimiento *uno* antes del procedimiento *dos* de la siguiente manera:

```
PROCEDURE uno (a : integer; VAR b : real); forward;
(* Se omite aquí el cuerpo del procedimiento uno *)
```

```
PROCEDURE dos (x : integer; VAR y : real);
declaraciones...
```

```
BEGIN
```

```
    proposiciones...
    uno (parámetros verdaderos);
    proposiciones...
```

```
END;
```

```
(* se omite la lista de parámetros, pero se agrega como comentario *)
(* por claridad. *)
```

```
PROCEDURE uno (* a: integer; VAR b: real *);
declaraciones...
```

```
BEGIN
```

```
    proposiciones...
    dos (parámetros verdaderos);
    proposiciones...
```

```
END;
```

Obsérvese que la palabra *forward* va seguida de un signo de punto y coma. Aunque no está reservada en Pascal estándar, debe evitarse el uso de la palabra *forward* para otros propósitos. Nótese también que la lista de parámetros del procedimiento *uno* llamado se omite en la declaración real del procedimiento, puesto que ya se incluyó en la declaración adelantada. A pesar de ello, es una buena costumbre de programación, como se acaba de mostrar, incluir la lista de parámetros en forma de comentario en la declaración del procedimiento que se llama.

La declaración adelantada también se puede aplicar a las funciones. He aquí un ejemplo de una declaración de este tipo:

```
FUNCTION filtro (a: char): char; forward;
```

## Transportabilidad

En este texto se ha hecho hincapié en el uso de Pascal estándar. La razón principal es que los programas escritos en Pascal estándar se pueden transportar de un sistema de cómputo a otro con un mínimo de cambios. Los programas que tienen esta característica se denominan *transportables*. Un programa transportable desarrollado, probado y depurado en un sistema de cómputo requerirá muy poca prueba y depuración adicional cuando se le cambie a otro sistema de cómputo.

Un problema que no se resuelve completamente al adherirse a Pascal estándar es el de las diferencias en los conjuntos de caracteres. Muchos sistemas de cómputo emplean el conjunto de caracteres ASCII, pero los sistemas IBM grandes utilizan el conjunto de caracteres EBCDIC, a la vez que otros fabricantes han definido conjuntos de caracteres adicionales. El problema principal es que pueden existir diferencias en las posiciones relativas de un carácter en los distintos conjuntos de caracteres. Esto puede producir resultados diferentes en las comparaciones de caracteres en dos máquinas distintas. Por ejemplo, el valor ‘A’ < ‘0’ es *true* cuando se evalúa en computadoras IBM grandes, pero en muchos otros sistemas (entre ellos las computadoras personales IBM y Apple) esta expresión es falsa. No existe una solución de aceptación universal a este problema. Muchas soluciones se basan en la traducción de un conjunto de caracteres a otro.

Si se emplean las extensiones con que cuenta un sistema de Pascal determinado, ya sea porque simplifican la resolución de un problema o porque mejoran su eficiencia, por lo menos deben identificarse claramente mediante comentarios las partes del programa en las que se hayan empleado características no estándar. Así, cuando se pase el programa a una computadora diferente, se podrán identificar con facilidad las proposiciones que tal vez requieran modificaciones.

He aquí una lista de algunos recordatorios importantes de Pascal relacionados con procedimientos y funciones.

### RECORDATORIOS DE PASCAL

- Los procedimientos o funciones deben declarar físicamente antes de invocarlos (a menos que se use una declaración adelantada).
- Los parámetros formales y verdaderos deben tener el mismo orden y tipo de datos.
- El parámetro verdadero que corresponda a un parámetro VAR formal debe ser una variable.
- Los parámetros de valor pueden ser variables, constantes o expresiones.
- Los parámetros de valor sirven como datos de entrada para un procedimiento o función.
- La palabra reservada VAR debe preceder al parámetro variable para cada tipo de datos en la lista de parámetros formales:

```
procedure ej (VAR n: real; VAR car1, car2: char);
```

- No conviene hacer referencia a las variables globales directamente desde el interior de un procedimiento o función.
- Es conveniente que las variables y constantes que se utilizan únicamente en un procedimiento o función sean variables locales.

- Un identificador local que tiene el mismo nombre que un identificador global tiene prioridad dentro del procedimiento o función.
- No se puede hacer referencia a los identificadores locales fuera de su alcance.
- La función booleana *eoln* produce el resultado *true* si el siguiente carácter que se va a leer es el carácter de fin de linea (<eoln>).
- La función booleana *eof* produce el resultado *true* si el siguiente carácter es el fin de archivo (<eof>).
- El tipo de datos del resultado de la función debe incluirse en el encabezado de la función:

**FUNCTION** potencia (base, exponente: real): real;

- El nombre de la función debe aparecer en el lado izquierdo de por lo menos una proposición de asignación de la función.
- El nombre de la función no debe aparecer en una expresión dentro de la función, a menos que se esté invocando como función recursiva.
- Las invocaciones de función no son válidas cuando se escriben como invocaciones de procedimientos. Deben aparecer como componentes de expresiones.

potencia (2.0, 3.0) (\* **no válida** \*)

x := potencia (3.0, 3.0) (\* **válida** \*)

- Las funciones se usan cuando se desea obtener únicamente un valor; en otros casos se emplean procedimientos.

## SECCIÓN 7.6 REPASO DEL CAPÍTULO

En este capítulo se analizaron los procedimientos y las funciones. Los parámetros de entrada de los procedimientos y funciones se denominan *parámetros de valor*. Si se emplea una variable únicamente para salida o tanto para entrada como para salida, se debe definir como parámetro variable (VAR) en la lista de parámetros. Tanto los procedimientos como las funciones se pueden anidar. Las reglas de alcance se aplican al programa principal, procedimiento, funciones e identificadores locales y globales. El alcance de un identificador es aquella parte del programa en la que se conoce el identificador. Examínense los siguientes procedimientos anidados:

```
PROCEDURE externo;
```

```
VAR a : integer;
```

```
PROCEDURE interno;
```

```
VAR b : integer;
```

```
PROCEDURE profundo;
```

```
VAR a : integer; ...
```

```
...
```

```
...
```

Los procedimientos *interno* y *externo* pueden hacer referencia a las variables declaradas en los procedimientos más grandes que los contienen. No es posible hacer referencia a las variables locales que se declaran en los procedimientos contenidos desde afuera de esos procedimientos, y dichas variables tienen prioridad sobre los mismos nombres de variable si se usan en los procedimientos más grandes que contienen a estos procedimientos. Las mismas reglas de alcance se aplican a los nombres de procedimientos.

En este capítulo se analizaron dos tipos de funciones: estándar y definidas por el usuario. Las funciones estándar son funciones predefinidas con que cuentan todas las versiones de Pascal. En este capítulo se habló de las funciones estándar *odd*, *eoln* y *eof*. La función *odd* prueba si un entero es par o non. Las funciones *eoln* y *eof* prueban si el siguiente carácter en un archivo de texto es el carácter de fin de línea (*<eoln>*) o el carácter de fin de archivo (*<eof>*).

Las funciones definidas por el usuario son las que escribe el programador y son muy parecidas a los procedimientos, con una excepción importante: las funciones devuelven exactamente un valor al punto de invocación. Se presentaron muchos ejemplos de funciones. En la sección 7.3 se hizo un análisis de las funciones recursivas, que son funciones que se pueden invocar a sí mismas. Las funciones recursivas se pueden emplear con naturalidad cuando la solución del problema tiene una definición recursiva.

Es este capítulo se presentó una aplicación de procedimientos y funciones a un problema de juego de multiplicación. Se incluyó un análisis de la generación de números aleatorios. El capítulo terminó con un análisis de los efectos secundarios y las declaraciones adelantadas que permiten cambiar el orden de las declaraciones.

En seguida se presenta un resumen de Pascal que se explicó en este capítulo, el cual puede usarse como referencia en el futuro.

## REFERENCIAS DE PASCAL

### 1 Parámetros

1.1 De valor: parámetro de entrada para procedimientos o funciones.

1.2 Variable: parámetros de salida, o de entrada y salida.

Ejemplo:

**PROCEDURE** (núm1: integer; VAR calif: integer);



parámetro de valor



parámetro variable

### 2 Procedimientos anidados: Procedimientos que se escriben dentro de otros procedimientos.

Ejemplo:

**PROCEDURE** principal (parámetros);  
declaraciones

PROCEDURE adentro (parámetros);  
declaraciones

```
PROCEDURE másinterno (parámetros);
declaraciones
BEGIN
    proposiciones
END (* másinterno *)
```

```
BEGIN
    proposiciones
END (* adentro *)
```

```
BEGIN
    proposiciones
END (* principal *)
```

El alcance de los identificadores es el cuadro indicado, salvo en los cuadros anidados donde se vuelven a declarar los identificadores.

### 3 Funciones.

#### 3.1 Funciones estándar: Predefinidas en Pascal estándar.

Funciones booleanas:

- odd*: Prueba si un número entero es non
- eoln*: Prueba la presencia del <eoln>
- eof*: Prueba la presencia del <eof>

#### 3.2 Funciones definidas por el usuario: escritas por el programador (similares a los procedimientos).

Ejemplo:

```
FUNCTION cotangente (ángulo: real): real;
BEGIN
    cotangente := (ángulo) / sen (ángulo)
END;
```

El nombre de la función debe aparecer por lo menos una vez en el lado izquierdo de una proposición de asignación. El tipo de datos del resultado se debe especificar en el encabezado de la función.

#### 3.3 Funciones recursivas: Funciones que se invocan a sí mismas, ya sea directa o indirectamente.

Ejemplo:

```

FUNCTION total (n: integer): integer;
BEGIN
  IF n = 1
    THE total := 1
  ELSE total := n + total (n - 1)
END;

```

- 4 Declaraciones adelantadas:** Permiten que el nombre de una función o procedimiento aparezca, y por tanto sea capaz de ser invocado, antes de su definición.

Ejemplo:

```
PROCEDURE último (número: integer); forward;
```

## Avance del capítulo 8

En el siguiente capítulo se examinarán con mayor detalle los tipos de datos. En particular, se analizará una característica atractiva del Pascal, los tipos de datos definidos por el usuario. Estos tipos de datos (conocidos como tipos enumerados) son los que define el usuario y pueden mejorar el diseño y la legibilidad de los programas. Además, se verán los tipos de datos restringidos, llamados *tipos de datos de subescala*, y algunas de sus aplicaciones. Las declaraciones de tipos que se estudiarán en el siguiente capítulo serán muy útiles durante el resto del curso cuando se estudien estructuras de datos más complicadas.

## Palabras clave del capítulo 7

alcance	identificador local
declaración adelantada	parámetro
efecto secundario	parámetro de valor
forward	parámetro variable
función definida por el usuario	parámetro formal
función <i>eof</i>	parámetro verdadero
función <i>eoln</i>	PROCEDURE
función estándar	número aleatorio
función <i>odd</i>	número seudoaleatorio
función recursiva	semilla
FUNCTION	transportabilidad
identificador global	

## ★ EJERCICIOS ESENCIALES

- 1 ¿Cómo se las arreglaría el lector para contar el número de veces que se invocó un determinado procedimiento o función en un programa en Pascal? ¿Qué tan fácil es hacerlo *a posteriori*, es decir, después de haber escrito el programa?
- 2 Examínense las siguientes proposiciones de procedimientos, donde *t1* y *t2* son identificadores de tipo únicos.
  - 1) PROCEDURE *p* (VAR *f1* : *t1*; *f2* : *t2*);
  - 2) PROCEDURE *p* (VAR *f1* : *t1*; VAR *f2* : *t2*);
  - 3) PROCEDURE *p* (*f1* : *t1*; *f2* : *t2*);

Para cada una de las siguientes invocaciones del procedimiento *p*, determine cuáles de las proposiciones de procedimientos anteriores se pueden usar. Supóngase que *v1* y *v2* son variables de los tipos *t1* y *t2*, respectivamente, *e1* y *e2* son expresiones de los tipos *t1* y *t2*, respectivamente.

- a)* *p* (*v1*, *v2*);      *c)* *p* (*e1*, *e2*);  
*b)* *p* (*v1*, *e2*);      *d)* *p* (*e1*, *v2*);
- 3 En el esqueleto de programa que se muestra en seguida (deliberadamente sin la sangría usual), dibújense cuadros para indicar el alcance de los identificadores.

PROGRAM cuadros (input, output);

PROCEDURE A;

PROCEDURE B;

END;

PROCEDURE C;

PROCEDURE D;

END;

END;

END;

PROCEDURE E;

FUNCTION F;

END;

END;

END.

- 4 En el esqueleto de programa que se muestra a continuación se incluyen definiciones y declaraciones CONST y VAR abreviadas. Después de cada uno de los tres comentarios que indican la colocación de las proposiciones de los procedimientos y el programa, indíquese cuáles identificadores son ace-

sibles, lo que representan (constantes o variables) y dónde se declararon (programa *x*, procedimiento *y* o *z*).

```
PROGRAM x;
CONST
    a, b, c;
VAR
    d, e, f;
PROCEDURE y;
CONST
VAR
    b, e;
PROCEDURE z;
CONST
VAR
    f, g;
BEGIN
    (* proposiciones del procedimiento z *)

```

```
END;
BEGIN
```

(\* proposiciones del procedimiento y \*)

```
END;
BEGIN
```

(\* proposiciones del programa x \*)

```
END.
```

- 5 Para cada uno de los siguientes problemas, indíquese si la forma más apropiada de expresar la solución es como procedimiento o como función.
- Dado un número primo, determinar el número primo inmediato más grande.
  - Dado un entero positivo  $N$  diferente de cero, determinar el número de factores únicos de  $N$  y los dos (o menos) factores únicos más grandes.
  - Dado un carácter del conjunto de caracteres ASCII, determinar su posición ordinal en el conjunto de caracteres EBCDIC.
  - Dado un número de años  $N$  y una cantidad que se va a obtener,  $C$ , determinar la tasa de interés anual necesaria para alcanzar  $C$  en  $N$  años si el interés es compuesto anual, y la tasa de interés anual necesaria para alcanzar  $C$  en  $N$  años si el interés es compuesto mensual.

### ★ ★ EJERCICIOS IMPORTANTES

- 6 Algunos lenguajes permiten la compilación externa de procedimientos y funciones. Es decir, los procedimientos y las funciones se pueden compilar de

- 7 Sugiérase un procedimiento para determinar el número de posiciones en un renglón de salida, que se requiere para representar correctamente un entero con signo. Por ejemplo, +403 requiere un mínimo de tres posiciones, pero -403 requiere cuatro. Recuérdese que cero requiere una posición en la salida, no cero posiciones.

### ★ ★ ★ EJERCICIOS ESTIMULANTES

- 8 La función de Ackermann se define de la siguiente manera, donde  $m$ ,  $n$  y el resultado son todos valores enteros:

$$\begin{aligned} A(m, n) &= n + 1 && \text{si } m = 0 \\ &= A(m - 1, 1) && \text{si } m \text{ no es cero y } n = 0 \\ &= A(m - 1, A(m, n - 1)) && \text{si ni } m \text{ ni } n \text{ son cero} \end{aligned}$$

Escríbase una función en Pascal que calcule el valor de la función de Ackermann, dados los parámetros  $m$  y  $n$ . (El lector puede optar por usar esta función en un programa de prueba, pero debe tener en cuenta que argumentos mayores de dos o tres para  $m$  y  $n$  ocasionan tiempos de ejecución extremadamente largos.)

## PROBLEMAS DEL CAPÍTULO 7 PARA RESOLUCIÓN EN COMPUTADORA

### ★ PROBLEMAS ESENCIALES

- 1 Escríbase una función booleana llamada *múltiplo* que tenga dos argumentos enteros  $m$  y  $n$ . *Múltiplo* debe producir el valor *true* si  $m$  es un múltiplo entero de  $n$  o  $n$  es un múltiplo entero de  $m$ . Escríbase un programa para probar esta función que lea una pareja de enteros de cada línea de datos de entrada, invoque a *múltiplo* y exhiba los enteros y el resultado de *múltiplo*.

Ejemplo de entrada:

4	7
4	8
16	8

Ejemplo de salida:

4	7	Falso
4	8	Verdadero
16	8	Verdadero

- 2 Escribase un procedimiento con tres parámetros variables reales  $a$ ,  $b$  y  $c$ . El efecto del procedimiento debe ser “girar” los valores de los parámetros hacia la derecha de manera que, después de la ejecución, el valor que originalmente estaba en  $a$  quede en  $b$ , el que estaba en  $b$  quede en  $c$  y el que estaba en  $c$  quede en  $a$ . Pruébese el procedimiento con un programa que lea tres números reales de cada línea de datos de entrada y exhiba los números reales después de la rotación.

Ejemplo de entrada:

4.7	1.0003	7.5
-12.5	6.5e-4	2.005e + 5

Ejemplo de salida:

7.5000e + 00	4.7000e + 00	1.0030e + 00
2.0050e + 05	-1.2500e + 01	6.5000e-04

- 3 Escribase una función llamada *inversión* que tenga tres parámetros. El primer parámetro, un valor real, especifica la tasa de interés anual al principio de cada año sobre inversiones que se dejaron en una institución financiera durante el año anterior. El segundo parámetro, también un valor real, especifica la inversión inicial en dólares. Suponiendo que los intereses se reinvierten al principio de cada año, la función debe dar como resultado el valor de la inversión después del número de años especificado por el tercer parámetro, un valor entero. Utilícese la iteración para determinar el valor de la inversión. Debe escribirse también un programa principal para probar la función *inversión*. El programa debe leer líneas que contienen tasas de interés, inversiones iniciales y plazos de inversión, invocar después la función *interés* para realizar el cálculo y exhibir los resultados.

Ejemplo de entrada:

0.12	1000.0	5
0.125	1000.0	5

Ejemplo de salida:

Tasa de interés = 0.120  
 Inversión = \$1000.00  
 Periodo = 5 años  
 Rendimiento = \$1762.34

Tasa de interés = 0.125  
 Inversión = \$1000.00  
 Periodo = 5 años  
 Rendimiento = \$1802.03

- 4 Escribase una función que capture un número octal de los datos de entrada y dé como resultado su valor en forma de entero. El número octal se presenta en la misma forma que un entero. Es decir, se deben pasar por alto los espacios a la izquierda y los caracteres de tabulación y fin de líneas, y en seguida leer dígitos octales (posiblemente precedidos de un signo de más o menos) y acumular el valor del número. Los números octales sólo pueden incluir los dígitos del cero al siete. Para determinar el valor de estos números se evalúa

$$d_i * 8^i + d_{i-1} * 8^{i-1} + \dots + d_1 * 8^1 + d_0 * 8^0$$

donde los dígitos del número octal, de izquierda a derecha, son  $d_i, d_{i-1}, \dots, d_1, d_0$ . Escribase también un programa principal que pruebe la función. Los datos de entrada deben tener únicamente un número octal por línea, y la salida debe ser el equivalente decimal del número.

Ejemplo de entrada:

$$\begin{array}{r} -701 \\ +42 \\ \hline 377 \end{array}$$

Ejemplo de salida:

$$\begin{array}{r} -449 \\ 34 \\ 255 \end{array}$$

- 5 Escribase un procedimiento que tenga como parámetro de valor único un valor entero. El procedimiento debe exhibir el valor del entero en el espacio mínimo requerido empleando únicamente proposiciones *write* con un parámetro de carácter. Esto se puede hacer si se trata la exhibición como problema recursivo. Supóngase que el entero que se va a exhibir se llama  $N$ . El problema se resolverá entonces mediante los siguientes pasos:

- 1) Si  $N$  es negativo, exhibase ‘–’ y conviértase  $N$  a  $-N$ .
- 2) Si  $N$  es mayor que o igual a 10, invóquese de manera recursiva este procedimiento con un argumento verdadero de  $N \text{ DIV } 10$
- 3) Exhibase  $N \text{ MOD } 10$

Escribase un programa principal para probar el procedimiento. Deberá leer valores enteros de los datos de entrada hasta que se llegue al fin de línea y exhibir los valores enteros (mediante el procedimiento) separados por una coma y un solo espacio. Puede hacerse la suposición de que esta salida cabe en un solo renglón.

Ejemplo de entrada

4    -0031    1000    315    -00099    0099

Ejemplo de salida:

4, -31, 1000, 315, -99, 99

### ★ ★ ★ PROBLEMAS ESTIMULANTES

- 6 Escribase una función que produzca el factor primo más pequeño de su argumento entero positivo. Un número primo  $N$  es divisible únicamente entre  $N$  y uno. Escribase un programa principal que use esta función para exhibir todos los factores primos de los enteros que aparecen en los datos de entrada; exhibanse los resultados para cada dato en un solo renglón de la salida, precedidos por el entero mismo y un signo de igual. Sepárense los factores primos mediante asteriscos.

Ejemplo de entrada:

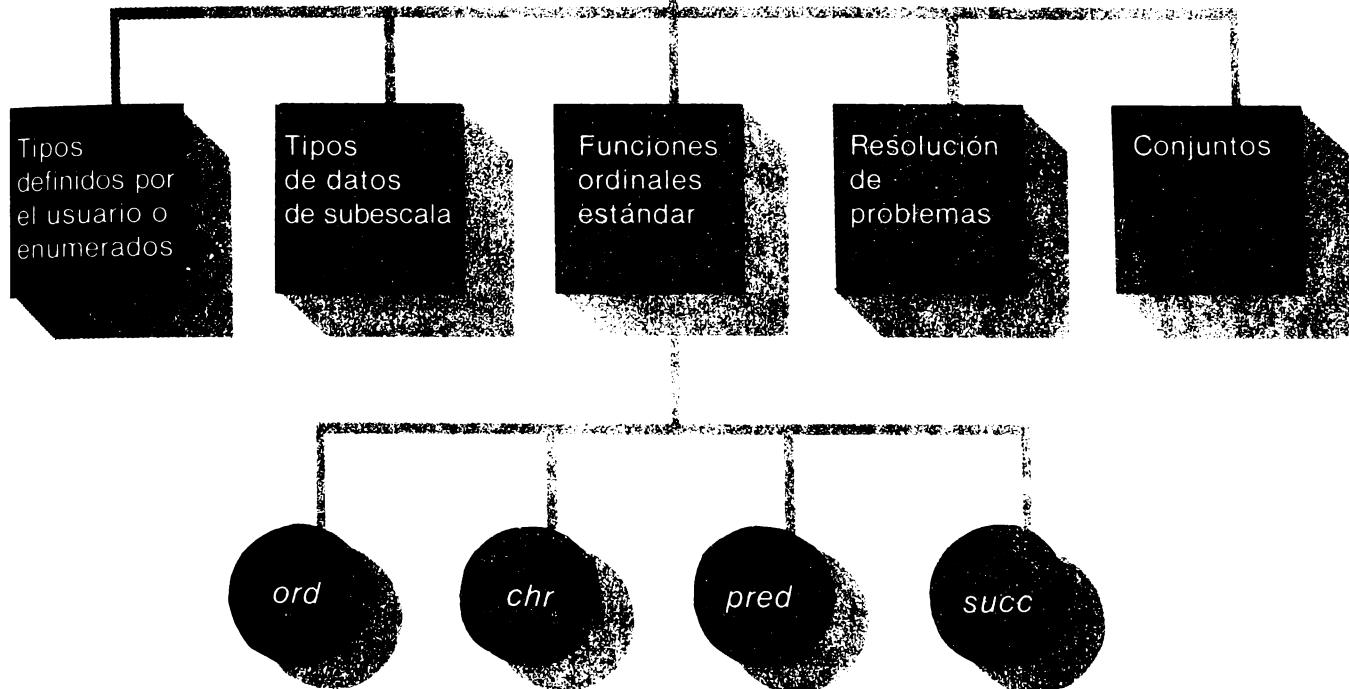
39    42    1517

Ejemplo de salida:

39 = 3 \* 13  
42 = 2 \* 3 \* 7  
1517 = 37 \* 41

# CAPÍTULO 8

## CAPÍTULO 8 TIPOS DE DATOS



## TIPOS DE DATOS

## OBJETIVOS

Después de completar este capítulo, el lector deberá ser capaz de:

- Declarar y aplicar tipos de datos definidos por el usuario (enumerados)
- Declarar y aplicar tipos de datos de subescala
- Reconocer y aplicar las funciones ordinales estándar: *ord*, *chr*, *pred* y *succ*
- Reconocer y aplicar el tipo de datos de conjunto
- Resolver, probar y depurar problemas que emplean tipos de datos enumerados

## PANORAMA GENERAL DEL CAPÍTULO

Una de las características atractivas del Pascal es la flexibilidad que permite en la definición de nuevos tipos de datos más apropiados para el problema que se va a resolver. En este capítulo se analizarán estos tipos de datos definidos por el usuario, o **tipos enumerados**. Por ejemplo, supóngase que se desea definir un nuevo tipo de datos llamado *color* cuyo valor puede ser cualquiera de las siguientes constantes:

rojo, blanco, azul, negro.

En Pascal se puede definir este tipo de datos mediante la siguiente definición de tipo (TYPE):

```
TYPE color = (rojo, blanco, azul, negro);
```

Después de esta proposición se pueden declarar variables cuyos valores pueden ser del tipo de datos *color*. Como se verá en este capítulo, los tipos de datos enumerados se pueden manipular de acuerdo a las mismas reglas que rigen a los tipos de datos estándar.

Dado un tipo de datos ordinal, como los enteros, de caracteres o booleanos (pero no reales), o un tipo enumerado, el Pascal permite restringir la escala de valores del tipo de datos. Estos tipos de datos restringidos se conocen como **tipos de datos de subescala**. Permiten la verificación automática de que los valores estén dentro de la escala especificada.

En este capítulo se incluye un análisis de las funciones ordinales estándar: *ord*, *chr*, *pred* y *succ*. Estas funciones son muy valiosas cuando se emplean tipos de datos ordinales. También se presenta una aplicación a la manipulación de caracteres.

Al final del capítulo se presenta una aplicación de los tipos de datos enumerados a un problema específico. Además, se incluye una introducción al tipo de datos de conjunto. La sección de técnicas de prueba y depuración examina el problema de compatibilidad de tipos cuando se usan tipos enumerados.

El Pascal permite al programador definir y declarar sus propios tipos de datos. Esta característica puede ser bastante útil cuando se traduce un algoritmo al código final en Pascal. En esta sección se estudiarán esos tipos de datos definidos por el usuario o enumerados.

Recuérdese que algunos tipos de datos simples, como los enteros, de caracteres y booleanos (pero no los reales) se llaman *ordinales* porque los valores (o constantes) que definen el tipo de datos están ordenados de manera precisa. Por ejemplo, las constantes booleanas están ordenadas de la siguiente manera.

`false < true`

Las constantes de caracteres también tienen un ordenamiento que incluye lo siguiente:

`'A' < 'B' < 'C' < 'D' < 'E' < ... < 'Z'`

Como se verá, los tipos de datos definidos por el usuario o enumerados también serán tipos ordinales.

### **Tipos de datos enumerados**

Los *tipos de datos definidos por el usuario* o *enumerados* son listas ordenadas de constantes a las que el usuario asigna un nombre. Por ejemplo, supóngase que se desea definir un tipo de datos llamado *vocal* que consta de las siguientes constantes ordenadas:

`A, E, I, O, U`

En este caso, el ordenamiento es el siguiente:

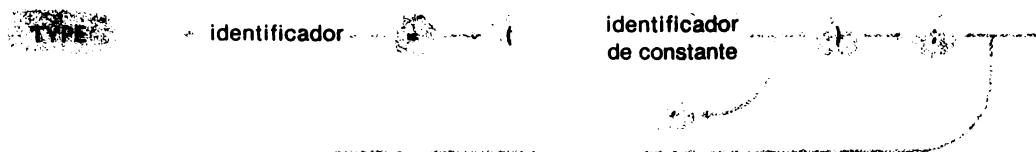
`A < E < I < O < U`

Estas constantes (*A, E, I, O* y *U*) *no* son las constantes de caracteres '`A`', '`E`', '`I`', '`O`', y '`U`'. En Pascal es posible declarar este tipo de datos ordinal mediante una *definición de tipo*:

`TYPE vocal = (A, E, I, O, U);`

Otro ejemplo que ilustra lo útil que puede ser un tipo de datos definido por el usuario es la siguiente definición de tipo para los días de la semana:

`TYPE día = (Domingo, Lunes, Martes, Miércoles,  
Jueves, Viernes, Sábado);`



**Figura 8-1** Diagrama de sintaxis de los tipos enumerados.

El diagrama de sintaxis de los tipos de datos enumerados se muestra en la figura 8-1.

En este punto es importante que el lector comprenda que la definición de tipo solamente define el tipo de datos ordinal. Para utilizar realmente el tipo de datos es preciso declarar variables del tipo dado. En Pascal, en la sección del programa dedicada a las declaraciones, las declaraciones de variables siguen a las definiciones de tipo. Considérense las siguientes declaraciones:

```
TYPE vocal = (A, E, I, O, U);
           día   = (Domingo, Lunes, Martes, Miércoles, Jueves,
                     Viernes, Sábado);
VAR letra : vocal;
    onomástico,
    díaferiado : día;
```

En este ejemplo, la variable *letra* es del tipo *vocal* y las variables *onomástico* y *díaferiado* son del tipo *día*. Existe otra forma de declarar estas variables que permite incluir la definición de tipo en la declaración de la variable. La siguiente declaración es equivalente a la anterior:

```
VAR letra : (A, E, I, O, U);
    onomástico,
    díaferiado: (Domingo, Lunes, Martes, Miércoles, Jueves, Viernes,
                  Sábado);
```

Se puede asignar a las variables cualquier valor constante especificado en la definición de tipo. Por ejemplo, las siguientes son proposiciones de asignación válidas:

```
letra := E;
onomástico := Domingo;
díaferiado := Sábado;
letra := A;
onomástico := díaferiado;
```

Las siguientes declaraciones de asignación *no son válidas*:

letra := 'A';	(* 'A' es del tipo de datos de carácter *)
letra := Lunes;	(* no se permite mezclar tipos de datos *)

Obsérvese en los ejemplos anteriores que no es posible mezclar diferentes tipos de datos definidos por el usuario. No obstante, puesto que los tipos de datos son ordinales, sí es posible comparar valores. Por ejemplo, la siguiente proposición IF es válida:

```
IF letra = U  
THEN writeln ('La vocal es U.')  
ELSE writeln ('La vocal no es U.')
```

Otro ejemplo de comparación de valores que aprovecha el ordenamiento es el siguiente:

```
IF (Domingo < díaferiado) AND (díaferiado < Sábado)  
THEN writeln ('El díaferiado cae entre semana.')  
ELSE writeln ('El díaferiado cae en fin de semana.')
```

Los tipos de datos enumerados se pueden manipular de la misma manera que los tipos de datos simples, con una excepción importante. En Pascal estándar *no es posible* leer o exhibir directamente las constantes de los tipos de datos enumerados. En el ejemplo anterior se declara la variable *letra* como del tipo *vocal*. Para exhibir el contenido de *letra* se puede usar una proposición CASE con el valor ordinal de *letra* como selector. Así, la siguiente proposición exhibiría el valor de *letra*:

```
CASE letra OF  
  A : writeln ('La vocal es A.');//  
  E : writeln ('La vocal es E.');//  
  I : writeln ('La vocal es I.');//  
  O : writeln ('La vocal es O.');//  
  U : writeln ('La vocal es U.');//  
END
```

### Problema 8.1

*Escribase un programa en Pascal que utilice tipos de datos enumerados para exhibir el número de días en cada mes (de un año no bisiesto) desde enero hasta diciembre, en orden.*

El siguiente programa en Pascal exhibirá los resultados deseados. Obsérvese el uso de las proposiciones FOR y CASE. Una de las ventajas de emplear un tipo enumerado es que mejora la legibilidad del programa.

**PROGRAMA díasenmes (input, output);**  
**(\* programa para exhibir el número de días en \*)**

(\* cada mes de un año no bisiesto \*)  
 TYPE tipomes = (Ene, Feb, Mar, Abr, May, Jun,  
                  Jul, Ago, Sep, Oct, Nov, Dic);  
 VAR mes : tipomes;  
 BEGIN  
     FOR mes := Ene TO Dic DO  
         CASE mes OF  
             Ene, Mar, May, Jul,  
             Ago, Oct, Dic : writeln ('31 días.');//  
             Abr, Jun, Sep, Nov : writeln ('30 días.');//  
             Feb : writeln ('28 días.')  
         END  
 END.

### Definiciones de tipo

En los programas en Pascal, las definiciones de tipo se colocan entre las declaraciones CONST y VAR. Así, la estructura de un programa en Pascal adquiere ahora la siguiente forma:

```
PROGRAMA nombre (input, output);
CONST declaraciones;
TYPE definiciones;
VAR declaraciones;
PROCEDURE o FUNCTION declaraciones;
BEGIN
    proposición;
    proposición;
    .
    .
    .
    proposición
END.
```

Las definiciones de tipo también pueden aparecer dentro de un procedimiento o función como definiciones locales.

He aquí una lista de reglas importantes que se refieren a las definiciones de tipos:

- Las constantes empleadas en un tipo enumerado deben ser diferentes de las empleadas en cualquier otro tipo de dato. Por ejemplo, la siguiente definición de tipo *no* es válida porque las constantes pertenecen al tipo de datos de caracteres:

```
TYPE calif = ('A', 'B', 'C', 'D', 'F');
```

Sin embargo, la siguiente sí sería una definición de tipo válida, en tanto no se hayan definido antes las constantes:

- No puede aparecer la misma constante en dos o más definiciones de tipo diferentes. Por ejemplo, las siguientes definiciones no son válidas porque se usa la constante *Viernes* en ambas definiciones:

TYPE

laborable = (Lunes, Martes, Miércoles,  
Jueves, Viernes);  
finsemana= (Viernes, Sábado, Domingo);

- La definición de tipo únicamente define un tipo de datos enumerado. Para reservar realmente las localidades de memoria necesarias para almacenar un valor del tipo especificado, se precisa una declaración de variable. El nombre del tipo no debe aparecer nunca en la parte ejecutable del programa. Piénsese en lo que sucede en el caso de los tipos de datos estándar. Nunca se emplean los nombres de los tipos de datos estándar (integer, real, char y boolean) en la parte ejecutable de los programas. El hecho de hacerlo causaría un significado diferente del nombre y sin duda produciría confusión.
- Las definiciones de tipo se pueden incluir como parte de las declaraciones VAR. Por ejemplo, la siguiente definición

TYPE estado = (soltero, casado, divorciado);  
VAR civil : estado;

puede, de hecho, escribirse como una sola declaración VAR:

VAR civil : (soltero, casado, divorciado);

Sin embargo, esta forma de la definición no se puede usar en el encabezado de un procedimiento o función. Es decir, no se permiten definiciones de tipos en la lista de parámetros formales de un procedimiento o función. Por ejemplo, el siguiente encabezado de procedimiento *no* es válido:

PROCEDURE buscar  
(VAR civil: (soltero, casado, divorciado));

Para permitir a los parámetros de valor y de variable asumir valores de un tipo enumerado, es menester definir el tipo enumerado en forma global (con respecto a los procedimientos que hacen referencia al tipo enumerado) mediante una definición TYPE. Así, una forma aceptable de declarar un parámetro formal de un tipo de datos enumerado es la siguiente:

TYPE estado = (soltero, casado, divorciado);  
VAR declaraciones;  
PROCEDURE buscar (VAR civil : estado);

En este caso, el tipo de datos enumerado *estado* se define globalmente y se conoce en todos los procedimientos y funciones dentro del alcance de la definición. Las reglas de alcance para las definiciones de tipo son las mismas que se aplican a las constantes, variables, procedimientos y funciones.

## EJERCICIOS DE LA SECCIÓN 8.1

- 1 Defínase un tipo enumerado para las siguientes definiciones:
  - a) Estados norteamericanos cuyos nombres comiencen con la letra A.
  - b) Los parientes del lector.
  - c) Tipos de bebidas gaseosas.
  - d) Sabores de helados.
- 2 Determinese cuáles de las siguientes proposiciones son válidas dada la siguiente declaración:

```
TYPE color = (rojo, blanco, azul, púrpura);
VAR coloración : color;
```

- a) read (rojo);
 writeln (rojo);
- b) read (coloración);
 write (coloración);
- c) coloración := blanco;
 writeln (coloración);
- d) coloración := blanco;
 CASE coloración OF
 rojo : writeln ('rojo');
 blanco : writeln ('blanco');
 azul : writeln ('azul');
 púrpura : writeln ('púrpura')
 END;
- e) coloración := azul;
 CASE coloración OF
 rojo : writeln (rojo);
 blanco : writeln (blanco);
 azul : writeln (azul);
 púrpura : writeln (púrpura)
 END;
- f) IF coloración = azul
 THEN writeln ('azul')
 ELSE writeln ('no azul')

- 3 Determinese si las siguientes definiciones de tipo son válidas o no:
  - a) TYPE letra = ('X', 'Y', 'Z');
  - b) TYPE lenguaje = (Pascal, Fortran, Basic);
  - c) TYPE materias = (matemáticas, historia, computación, biología);
 carrera = (matemáticas, computación);

- d) TYPE estado = (residente, ciudadano, extranjero);  
     nacionalidad = (americano, europeo, africano, asiático, otra);
- e) TYPE código = (1, 2, 3, 4, 5);
- f) TYPE código = (c1, c2, c3, c4, c5);
- 4 Determíñese si las siguientes declaraciones son válidas o no.
- a) TYPE estado = (soltera, casada, comprometida, divorciada);  
     VAR type : estado;
- b) TYPE ciudad = (NuevaYork, LosÁngeles, Chicago, Houston);  
     VAR ciudad : type;
- c) VAR ciudad : (NuevaYork, Londres, París, Roma);
- d) PROCEDURE encontrar (VAR ciudad : (NuevaYork, Londres, Roma));
- e) TYPE trabajo = (obrero, oficinista, indefinido);  
     PROCEDURE buscar (VAR empleo: trabajo);

- 5 Estúdiese el siguiente programa.

```

PROGRAM bueno (input, output);
TYPE
    materia = (matemáticas, historia, computación, geografía,
                física);
VAR
    a, b : materia;
BEGIN
    a : = matemáticas;
    b : = computación;
    IF a > b
        THEN write ('magnífico')
        ELSE write ('excelente')
END.

```

¿Cuáles de las siguientes afirmaciones son ciertas?

- a) En la proposición IF,  $a$  y  $b$  no se pueden comparar porque no son números.
- b) El programa exhibirá “magnífico”.
- c) El programa exhibirá “excelente”.
- d) La proposición IF provocará un error de ejecución.
- e) Ninguna de las afirmaciones es cierta.

- 6 Determíñese la salida del siguiente programa:

```

PROGRAM prueba (input, output);
TYPE
    ciudad = (Boston, NuevaYork, Miami, Chicago);
VAR
    ciudad1, ciudad2 : ciudad;
PROCEDURE exhibir (ciudad1, ciudad2; ciudad);
BEGIN
    IF ((ciudad1 >= Boston) AND (ciudad1 <= Chicago)) AND
       ((ciudad2 >= Boston) AND (ciudad2 <= Chicago))

```

```

THEN IF ciudadA > ciudadB
|   THEN CASE ciudadA OF
|       Boston: writeln ('Viaje a Boston.');
|       NuevaYork: writeln ('Viaje a Nueva York.');
|       Miami: writeln ('Viaje a Miami.');
|       Chicago: writeln ('Viaje a Chicago.')
|   END
|   ELSE writeln ('No se recomienda viajar.')
|   ELSE writeln ('Error en los datos.')
END;
(* Programa principal *)
BEGIN
    ciudad1 := Chicago;
    ciudad2 := NuevaYork;
    exhibir (ciudad1, ciudad2);
    ciudad1 := Boston;
    ciudad2 := Miami;
    exhibir (ciudad1, ciudad2)
END.

```

- 7 Escríbase un procedimiento en Pascal que emplee tipos enumerados y que exhiba el nombre del mes dado el nombre abreviado del mes como parámetro. Por ejemplo, si se pasa el nombre de mes Mar al procedimiento, se debe exhibir la palabra Marzo. Defínase el tipo enumerado globalmente.

## SECCIÓN 8.2 TIPOS DE DATOS DE SUBESCALA

Supóngase que se está escribiendo un programa en Pascal que calcula las calificaciones de examen de ciertos estudiantes, las cuales se encuentran en la escala de cero a 100. Se puede emplear una declaración de variable que restrinja la escala de los enteros a los números cero a 100 así:

**VAR califexamen : 0..100;**

La variable *califexamen* va seguida de un signo de dos puntos y la escala de valores, que comienza con el primer valor o el más pequeño (cero) y el último valor o el más grande (100). Ambos valores se separan por medio de *dos* puntos. En Pascal, los tipos de datos ordinales restringidos a una escala específica se llaman *tipos de datos de subescala*.

Las ventajas de emplear tipos de datos de subescala son las siguientes:

- 1 Se especifica explícitamente la escala de valores, en beneficio del diseño y legibilidad del programa.
- 2 Si el valor asignado a una variable declarada con un tipo de datos de subescala queda fuera de la escala especificada, la computadora proporcionará un mensaje diagnóstico apropiado.

Los tipos de subescala se pueden derivar de cualquier tipo ordinal estándar o definido por el usuario, con la excepción de los reales. Las siguientes declaraciones emplean tipos de subescala derivados de los tipos de datos estándar:

VAR escala: 1..10;	(* la escala incluye 1,2,3,4,5,6,7,8,9 y 10 *)
mayúscula: 'A'..'Z';	(* la escala incluye 'A', 'B', . . . , 'Z' *)
interna: -3..3;	(* la escala incluye -3, -2, -1,0,1,2,3 *)

Supóngase que se tiene la siguiente definición de tipo de datos enumerado:

TYPE día = (Dom, Lun, Mar, Mié, Jue, Vie, Sáb);

Entonces la siguiente será una declaración válida de tipo de datos de subescala:

VAR laborable: Lun..Vie; (\* la escala incluye Lun, Mar, Mié, Jue, Vie \*)

Si se trata de asignar a *laborable* un valor fuera de la escala válida, se producirá un error de ejecución.

He aquí algunas declaraciones válidas adicionales que usan subescalas de los tipos de datos estándar:

VAR

calif: 'A'..'F';	(* la escala incluye 'A', 'B', 'C', 'D', 'E', y 'F' *)
positivo: 1..máxint;	(* la escala incluye todos los números positivos hasta máxint *)
temperatura: -50..100;	(* la escala incluye todos los enteros desde -50 hasta 100 *)
turno: Mié..Sáb;	(* la escala incluye Mié, Jue, Vie y Sáb *)

Conviene tomar nota de unas cuantas reglas que se aplican al uso del tipo de datos de subescala:

- Los valores inferior y superior en la declaración de tipo de subescala deben ser del mismo tipo de datos. Éste se denomina tipo de datos *huésped*. El valor inferior debe ser menor que el valor superior o igual a él. Por tanto, estos ejemplos *no* son válidos:

VAR minúscula: 'z'..'a'; (\* orden erróneo \*)  
dígitos: 0..9; (\* diferente tipo de datos \*)

- Los valores de subescala incluyen todos los valores constantes entre los valores inferior y superior, inclusive. Por ejemplo, no es posible tener una subescala de los primeros cinco enteros pares.
- Se pueden combinar diferentes subescalas del mismo tipo en expresiones y proposiciones de asignación. Sin embargo, se producirá un error si algún valor queda fuera de la escala especificada. Considérense, por ejemplo, las siguientes declaraciones:

```
VAR prueba: 0..100;
      total: integer;
      escala: 1..10;
```

Las siguientes proposiciones de asignación son válidas, pero la verificación de escalas seguirá activa durante su ejecución:

```
escala := total DIV prueba;
total := escala * prueba;
```

En la sección de técnicas de prueba y depuración se examinará la compatibilidad de tipos con mayor detalle.

- Para emplear las declaraciones de tipo de subescala con parámetros formales en un encabezado de función o procedimiento, es preciso definir globalmente el tipo de subescala mediante una definición TYPE. Por ejemplo:

```
TYPE calif = 0..100;
VAR declaraciones:
PROCEDURE prueba (examen : calif);
```

En este caso, el parámetro de valor *examen* puede asumir valores del tipo de datos de subescala *calif* (escala de cero a 100). El siguiente encabezado de procedimiento *no es válido*:

```
PROCEDURE prueba (exam : 0..100);
```

- Los tipos de subescala se pueden definir en términos de constantes previamente declaradas. Por ejemplo, el siguiente segmento en Pascal utiliza las declaraciones de constantes en la definición de tipo:

```
PROGRAM ejemplo (input, output);
CONST bajo = 0;
      alto = 100;
TYPE calif = bajo..alto;
VAR prueba : calif;
```

El siguiente problema es un ejemplo del uso de tipos de datos de subescala.

### Problema 8.2

*Dos dados se tiran 500 veces. El resultado de cada tirada se capture como dato de entrada de un programa. Escríbase un programa en Pascal que cuente y exhiba el número de resultados que tienen un valor total de siete. Tómese nota de que la escala de valores para cada resultado es de dos a 12.*

El siguiente programa en Pascal resuelve este problema. Obsérvese el uso de tipos de datos de subescala.

```

PROGRAM siete (input, output);
(* Contar y exhibir el número de sietes que *)
(* resultan de tirar un par de dados. *)
CONST
    baja = 2;          (* tirada más baja posible *)
    alta = 12;         (* tirada más alta posible *)
    máx = 500;         (* número de tiradas *)
TYPE
    valor = baja..alta; (* datos de entrada permitidos *)
VAR
    índice : 1..máx;   (* cuenta tiradas *)
    tirada : valor;   (* valor de la tirada actual *)
    contador : 0..máx  (* cuenta tiradas de siete *)
BEGIN
    contador := 0;
    FOR índice := 1 TO máx DO
        BEGIN
            read (tirada);
            IF tirada = 7
                THEN contador := contador + 1;
        END;
        writeln ('Número de tiradas:', máx:1);
        writeln ('Número de sietes:', contador:1)
    END.

```

## EJERCICIOS DE LA SECCIÓN 8.2

- 1 Definir un tipo de subescala para cada una de las siguientes definiciones:
  - a) Los enteros no negativos.
  - b) Los enteros mayores de 100.
  - c) Los caracteres de la primera mitad del alfabeto.
  - d) Los caracteres de la segunda mitad del alfabeto.
- 2 Encuéntrese la escala de valores para la variable *tiempo* en la siguiente declaración:

```

CONST
    mín = 99;
    máx = 1000;
TYPE
    duración = mín..máx;
VAR
    tiempo : duración;

```

- 3 Examínense estas declaraciones:

```

TYPE
    tipodía = (Lunes, Martes, Miércoles, Jueves,
               Viernes, Sábado, Domingo);

```

## VAR

día : tipodía;  
 laborable : Lunes..Viernes;  
 finsemana : Sábado..Domingo;

- a) ¿Se producirá un error de ejecución si se asigna el valor Martes a *finsemana*?  
 b) ¿Se producirá un error de ejecución si se asigna el valor Viernes a *laborable*?

4 Determinese la escala de valores de todas las variables.

## TYPE

mes = (Ene, Feb, Mar, Abr, May, Jun);

## VAR

letra : 'A'..'Z';  
 negativo : - máxint..-1;  
 unosdígitos : '0'..'3';  
 temp : -32..32;  
 primavera : Mar..Jun;  
 semestre : Ene..Jun;

5 Supóngase que se hacen las siguientes declaraciones en un programa correcto en Pascal:

## VAR

bajo : 1..5;  
 medio : 1..10;  
 alto : 6..20;  
 grande : '1'..'10';

Si el valor de *alto* es siete, ¿cuál de estas asignaciones *no* generará un error de compilación o de ejecución? Todas las asignaciones son independientes.

- a) *bajo* := *alto*;  
 b) *alto* := 3 \* *alto*;  
 c) *grande* := *alto*;  
 d) *medio* := *alto*;  
 e) *medio* := *alto* - 7;

6 ¿Cuáles de las siguientes declaraciones de subescala son válidas?

- a) TYPE etiq1 = '0'..9;  
 b) TYPE etiq2 := -3..6;  
 c) TYPE etiq3 = -1..-3;  
 d) TYPE etiq4 = 0.0..9;  
 e) TYPE etiq5 = 1..1;

7 Escríbase un programa en Pascal con declaraciones de subescala para determinar si una secuencia de cinco dígitos capturados como datos de entrada constituyen un *palindroma numérico*. Un *palindroma numérico* es una secuencia de dígitos que se puede leer en ambos sentidos sin que exista diferencia. Por ejemplo, los números de cinco dígitos 12321, 11211, 54345 y 22222 son

### SECCIÓN 8.3 FUNCIONES ORDINALES ESTÁNDAR: *PRED, SUCC, CHR Y ORD*

Cuando se emplean tipos de datos ordinales, incluso tipos enumerados, muchas veces es necesario manipular de alguna forma los datos. Para facilitar esto, el Pascal cuenta con las siguientes cuatro funciones ordinales estándar (predefinidas): *pred*, *succ*, *chr* y *ord*. He aquí un breve resumen de las cuatro funciones, donde se da el tipo de datos que se requiere en el argumento y una descripción del resultado que se obtiene con cada una.

<i>función ordinal estándar</i>	<i>descripción</i>	<i>tipo de argumento</i>	<i>tipo de resultado</i>
<i>pred(x)</i>	Predecesor de <i>x</i>	Cualquier tipo ordinal	El mismo tipo
<i>succ(x)</i>	Sucesor de <i>x</i>	Cualquier tipo ordinal	El mismo tipo
<i>chr(x)</i>	Carácter cuyo código es <i>x</i>	Integer	Char
<i>ord(x)</i>	Posición de <i>x</i> en su tipo de datos	Cualquier tipo ordinal	Integer

Ahora se examinará con detenimiento cada una de estas funciones.

#### **Pred y Succ**

La función *pred(x)* produce el predecesor del argumento *x* en su tipo de datos ordinal. El predecesor es la constante que precede inmediatamente a *x* en el tipo de datos que contiene a *x*. En un tipo de datos ordinal, todas las constantes, salvo la primera, tienen un predecesor inmediato. Por ejemplo,

<i>pred('C')</i>	es igual a	'B'
<i>pred('8')</i>	es igual a	'7'
<i>pred(3)</i>	es igual a	2

Supóngase que se define el siguiente tipo de datos enumerado:

TYPE días = (Dom, Lun, Mar, Mié, Jue, Vie, Sáb);

Los siguientes ejemplos ilustran el efecto de aplicar la función *pred* a las constantes del tipo de datos enumerado:

<i>pred(Sáb)</i>	es igual a	Vie
<i>pred(Mié)</i>	es igual a	Mar
<i>pred(Lun)</i>	es igual a	Dom

El valor de “*pred(Dom)*” no está definido, ya que la constante *Dom* no tiene predecesor inmediato. Por la misma razón, “*pred(-máxint)*” no está definido.

La función *succ(x)* produce el sucesor del argumento *x* en su tipo de datos ordinal. El sucesor es la constante que sigue inmediatamente a *x* en el tipo de datos que contiene a *x*. Todas las constantes de un tipo de datos ordinal, con excepción de la última, tienen un sucesor inmediato. Por ejemplo,

<i>succ('a')</i>	es igual a	'b'
<i>succ(0)</i>	es igual a	1
<i>succ(-3)</i>	es igual a	-2
<i>succ(false)</i>	es igual a	true
<i>succ('X')</i>	es igual a	'Y'
<i>succ(Dom)</i>	es igual a	Lun

Tómese nota de que “*succ(Sáb)*” no está definida para el tipo enumerado definido con anterioridad. De manera similar, “*succ(máxint)*” no está definido, ya que el entero *máxint* no tiene sucesor inmediato en Pascal.

El argumento y resultado de ambas funciones, *pred* y *succ*, puede ser de cualquier tipo de datos ordinal, incluso los tipos enumerados. La siguiente tabla de ejemplos adicionales de las funciones *pred* y *succ*.

<i>ejemplo</i>	<i>resultado</i>
<i>pred(10)</i>	9
<i>pred('1')</i>	'0'
<i>pred(true)</i>	false
<i>pred(false)</i>	no definido
<i>succ(pred(' '))</i>	' '
<i>pred(succ(2))</i>	2
<i>succ(succ(8))</i>	10

### Ord y Chr

Las constantes de un tipo de datos ordinal están ordenadas según su posición en el tipo de datos. La función *ord(x)* produce la posición ordinal del argumento *x* en su tipo de datos. Es decir, *ord(x)* produce un entero que indica la posición de *x* en el ordenamiento de constantes del mismo tipo de datos que tiene *x*. Considérese la siguiente definición de tipo enumerado:

posic. ordinal =	0      1      2      3      4      5      6      7      8      9      10      11
	$\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$
TYPE mes =	(Ene,Feb,Mar,Abr,May,Jun,Jul,Ago, Sep,Oct, Nov,Dic);

Obsérvese que la posición ordinal de la primera constante en todos los tipos de datos enumerados es cero, *no* uno. Así, se tendrán los siguientes resultados:

<i>ord(Feb)</i>	es igual a	1
<i>ord(Dic)</i>	es igual a	11

En todos los tipos de datos enumerados la posición ordinal de la primera constante del tipo es cero. Esto incluye al tipo de datos booleano, de manera que “`ord(false)`” es cero. La posición ordinal de cualquier entero es igual al entero mismo. Así, “`ord(8)`” es igual a 8 y “`ord(-6)`” es igual a -6. La función `ord` puede aplicarse a un valor de cualquier tipo ordinal y siempre produce un resultado entero. En particular, se puede utilizar la función `ord` para convertir valores del tipo de dato de carácter en valores enteros. Recuérdese que la computadora representa internamente los caracteres mediante valores enteros. Los valores que se usen para esta representación dependen del sistema de cómputo de que se trate y del conjunto de caracteres que se utilice. Por ejemplo, muchos sistemas de cómputo, entre ellos varios de los fabricados por Digital Equipment Corporation y la mayor parte de las computadoras personales, emplean el código ASCII. Otros, como las máquinas grandes fabricadas por IBM, emplean el código EBCDIC. El apéndice F proporciona los códigos empleados en ambos esquemas.

Si un sistema de cómputo emplea el código ASCII se obtendrán los siguientes resultados con la función `ord`:

<code>ord('A')</code>	es igual a	65
<code>ord('B')</code>	es igual a	66
<code>ord('C')</code>	es igual a	67
<code>ord('0')</code>	es igual a	48
<code>ord('1')</code>	es igual a	49

La función `chr(x)` es el inverso de la función `ord` aplicada a argumentos de caracteres. Es decir, si  $x$  es un entero que representa una posición ordinal en el conjunto de caracteres, entonces `chr(x)` será el carácter que está en esa posición. Se recomienda comparar los siguientes ejemplos con los que se acaban de dar. Una vez más, se supone el empleo del código ASCII.

<code>chr(65)</code>	es igual a	'A'
<code>chr(66)</code>	es igual a	'B'
<code>chr(67)</code>	es igual a	'C'
<code>chr(48)</code>	es igual a	'0'
<code>chr(49)</code>	es igual a	'1'
<code>chr(ord('A'))</code>	es igual a	'A'

En seguida se examinará un programa que exhibe las posiciones ordinales y los caracteres correspondientes para un conjunto de caracteres dado con  $N$  (entero positivo) caracteres.

### Problema 8.3

*Una determinada computadora tiene  $N$  caracteres en su conjunto de caracteres. Escribáse un programa en Pascal con el propósito de exhibir las posiciones or-*

dinales y los caracteres en esas posiciones. Supóngase que  $N$  no puede ser mayor de 512.

El siguiente programa supone que el número de caracteres que tiene el conjunto de caracteres,  $N$ , se proporcionará como dato de entrada (su valor se almacenará en la variable *núm*). Se exhiben los caracteres y sus posiciones ordinales. Tómese nota del límite superior de 512 para todos los tipos de subescala.

```

PROGRAM exhibir (input, output);
(* Exhibir los caracteres con valores ordinales *)
(* en la escala de uno a N. N es un dato de entrada *)
VAR
    núm : 1..512;      (* número de caracteres *)
    índice : 0..511;    (* posición ordinal *)
BEGIN
    write ('Escríbase el número de caracteres en el conjunto:');
    readln (núm);
    FOR índice := 0 TO núm-1 DO
        writeln ('Posición ordinal:', índice:1, ':5';
                  'Carácter:', chr(índice):1)
END.

```

En el conjunto de caracteres ASCII (y en todos los conjuntos de caracteres que se emplean con Pascal) los dígitos están ordenados en forma contigua. Es decir, los dígitos están en secuencia ascendente sin otros caracteres entre ellos. La siguiente tabla muestra el orden de los dígitos y sus posiciones ordinales en el conjunto de caracteres ASCII.

<i>dígito</i>	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
<i>ordinal</i>	48	49	50	51	52	53	54	55	56	57

En ocasiones puede ser necesario convertir la representación como carácter de un dígito decimal en su equivalente entero. Por ejemplo, ¿cómo convertir el carácter '5' en su equivalente entero 5? El valor entero de '5' es igual a su distancia desde '0' en todos los conjuntos de caracteres que se emplean en Pascal, por lo que puede verse, por ejemplo, que

$$\begin{aligned} \text{ord('5')} - \text{ord('0')} &= 53 - 48 = 5 \\ \text{ord('6')} - \text{ord('0')} &= 54 - 48 = 6 \\ \text{ord('0')} - \text{ord('0')} &= 48 - 48 = 0 \end{aligned}$$

Así, puede verse que para convertir cualquier variable de carácter arbitraria que contenga un dígito decimal (la llamaremos *dígito*) en su equivalente entero, basta con usar la expresión

$$\text{ord(dígito)} - \text{ord('0')}.$$

puesto que los dígitos son contiguos en todos los conjuntos de caracteres que se pueden usar con el Pascal, esta fórmula funcionará en cualquier programa. Aun-

que el valor de “ord(‘0’)” puede variar (según el sistema de cómputo que se emplee), el compilador de Pascal para ese sistema de cómputo hará que se produzca el valor correcto. Basta con restar ese valor de “ord(dígito)” para obtener el equivalente entero.

Se puede invertir el proceso ilustrado cuando se desea obtener el equivalente como carácter del valor de una variable entera que contiene un valor en la escala cero a nueve. Obsérvese el resultado de las siguientes expresiones:

```
chr(0 + ord('0')) = '0'  
chr(1 + ord('0')) = '1'
```

1

1

`chr(9 + ord('0')) = '9'`

De manera que la expresión

```
chr(val + ord('0'))
```

sirve para convertir de un valor entero *val* en la escala de cero a nueve al valor de carácter equivalente.

Ahora se aplicarán las funciones ordinales a un problema que lee un entero no negativo de una cadena de caracteres y lo convierte en el valor numérico verdadero.

### Problema 8.4

*Escríbase un procedimiento en Pascal que lea una secuencia de caracteres, pasándolos por alto hasta que se encuentra el primer dígito decimal. En ese momento el procedimiento deberá convertir el dígito decimal y cualesquiera dígitos que le sigan inmediatamente (hasta llegar al primer carácter que no sea dígito) en el equivalente entero. Este valor entero se debe asignar al único parámetro variable entero del procedimiento.*

El siguiente procedimiento, llamado *convertir*, resuelve el problema. Tómese nota de que cada vez que se llega a un dígito nuevo consecutivo se multiplica el resultado anterior por 10 y se le suma el valor del dígito nuevo. La estrategia usada es consecuencia directa de la propiedad de que un número decimal, por ejemplo el número de tres dígitos  $d_1d_2d_3$  es igual a  $(d_1 * 10 + d_2) * 10 + d_3$ . Por ejemplo,  $471 = (4 * 10 + 7) * 10 + 1$ .

BEGIN

```

núm := 0;
REPEAT                                     (* buscar el primer dígito *)
    read (uncar)
UNTIL (uncar >= '0') AND (uncar = '9');
REPEAT                                     (* convertirlo en entero *)
    núm := núm * 10 + ord (uncar) - ord('0');
    read (uncar);                         (* capturar el siguiente carácter *)
UNTIL (uncar < '0') OR (uncar > '9')
END;
```

Este procedimiento es muy útil en programas que manipulan cadenas de caracteres, como se verá en un capítulo posterior. Como ejercicio, pruébese el procedimiento con la siguiente cadena como datos de entrada:

Blaise Pascal nació en 1623.

## EJERCICIOS DE LA SECCIÓN 8.3

1 Determíñese el resultado de estas expresiones:

- a) pred (8)                  b) succ (1)
- c) succ(-3)                d) pred ('8')
- e) succ (true)              f) pred (- máxint)

2 Determíñese el resultado de estas expresiones:

- a) pred (pred (0))                  b) pred (succ (100))
- c) succ (pred ('0'))                d) succ (succ ('2'))
- e) pred (pred (pred (9)))         f) succ (pred (succ ('A')))

3 Examíñese la siguiente definición:

### TYPE

```
raro = (gugol, nudol, brudol, cudol, zudol, budol);
```

Determíñese el valor de estas expresiones:

- a) ord (gugol)                  b) ord (zudol)
- c) succ (brudol)                d) succ (budol)
- e) pred (gugol)                f) ord (succ (zudol))

4 Obténgase el valor de las siguientes expresiones:

- a) ord ('7') - ord ('0')
- b) ord ('1') - ord ('1')
- c) chr (3 + ord ('0'))
- d) chr (0 + ord ('0'))

5 Utilícese el procedimiento *convertir* de esta sección con las siguientes cadenas como datos de entrada:

- a) El año 2001 está próximo.
- b) \$37.00 es la cantidad.

TYPE

vocal (a, e, i, o, u);

VAR

letra : vocal;

uncar : char;

¿Se ejecutará sin error el siguiente código en Pascal?

```
letra := a;
WHILE letra <= u DO
BEGIN
    read (uncar);
    writeln ('El carácter capturado es', uncar);
    letra := succ (letra)
END
```

- 7 Si se emplea la declaración del problema 6, ¿se ejecutará sin error el siguiente código?

```
letra := u;
REPEAT
    read (uncar);
    writeln ('El carácter capturado es', uncar);
    letra := pred (letra)
UNTIL letra = a
```

- 8 Vuélvase a escribir el procedimiento *convertir* de esta sección con ciclos WHILE en vez de ciclos REPEAT-UNTIL.

## SECCIÓN 8.4 RESOLUCIÓN DE PROBLEMAS CON TIPOS ENUMERADOS

En esta sección se resolverá el siguiente problema mediante un tipo de datos enumerados, lo cual hará más legible el programa en Pascal.

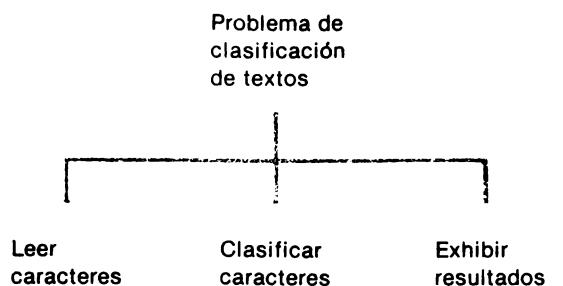
Un profesor de español necesita un programa de clasificación de textos que clasifique todos los caracteres de un enunciado como vocales mayúsculas, consonantes mayúsculas o caracteres especiales, e informe del número total de caracteres del enunciado, así como el porcentaje de cada tipo de carácter capturado. El profesor supone que el enunciado en que aparecen los caracteres termina con un punto y que los únicos caracteres especiales (aparte del punto) que se pueden incluir son espacios en blanco, comas, y signos de punto y coma y de dos puntos.

Este problema se puede subdividir en los tres subproblemas siguientes (véase la Fig. 8-2):

SUBPROBLEMA 1: Leer los caracteres y actualizar el número total de caracteres leídos.

SUBPROBLEMA 2: Clasificar cada carácter y actualizar el contador apropiado.

SUBPROBLEMA 3: Exhibir el número total de caracteres y calcular y exhibir los porcentajes.



**Figura 8-2** Diseño descendente del problema de clasificación.

He aquí un algoritmo detallado:

*Algoritmo de clasificación de textos*

Asignar el valor inicial cero a todos los contadores.  
 Leer un carácter.  
 Mientras el carácter no es punto:  
     Actualizar el contador de total de caracteres.  
     Clasificar el carácter.  
     Actualizar el contador apropiado.  
     Leer otro carácter.  
 Exhibir el total y los porcentajes.

A continuación se muestra el programa en Pascal. Adviéntase el uso del tipo de datos enumerado “clase = (vocal, consonante, especial)”.

```

PROGRAM clasetexto (input, output);
(* Programa para clasificar los caracteres de un enunciado como *)
(* vocal, consonante o carácter especial y exhibir el total de      *)
(* caracteres, así como los porcentajes.                            *)
TYPE
  clase = (vocal, consonante, especial);
VAR
  uncar : char;                      (* carácter capturado *)
  total,                                (* número total de caracteres *)
  cuentav,                               (* número de vocales *)
  cuentac,                               (* número de consonantes *)
  cuentae : 0..máxint;                 (* número de caracteres especiales *)
  categoría : clase;                  (* tipo de carácter *)
BEGIN
  total := 0;                          (* valor inicial de la cuenta total *)
  cuentav := 0;                         (* valor inicial del contador de vocales *)

```

```

cuentac := 0;      (* valor inicial, contador de consonantes *)
cuentae := 0;      (* valor inicial, contador car. especiales *)
read (uncar);
WHILE uncar <> '.' DO
BEGIN
    (* Aumentar contador de caracteres totales *)
    total := total + 1;
    (* determinar clase del carácter *)
    IF (uncar = 'A') OR (uncar = 'E') OR
        (uncar = 'I') OR (uncar = 'O') OR
        (uncar = 'U')
    THEN categoría := vocal
    ELSE IF (uncar = ' ') OR (uncar = ',') OR
        (uncar = ':') OR (uncar = ';')
    THEN categoría := especial
    ELSE categoría := consonante;
    (* Aumentar el contador de la categoría apropiada *)
    CASE categoría OF
        vocal : cuentav := cuentav + 1;
        consonante : cuentac := cuentac + 1;
        especial : cuentae := cuentae + 1;
    END (* del case *)
    read (uncar)  (* capturar el siguiente carácter *)
END; (* del while *)
(* exhibir resultados *)
writeln ('El total de caracteres es', total:1, '.');
writeln (cuentav / total * 100 : 6 : 2,
        '% fueron vocales.');
writeln (cuentac / total * 100 : 6 : 2,
        '% fueron consonantes.');
writeln (cuentae / total * 100 : 6 : 2,
        '% fueron caracteres especiales.');
END.

```

---

*Programa clasetexto*

-----  
 Datos de entrada: **SER O NO SER.**  
 Salida: El total de caracteres es 12.  
 33.33 % fueron vocales.  
 41.66 % fueron consonantes.  
 25.00 % fueron caracteres especiales.

-----  
*Programa clasetexto: ejemplo de ejecución*

En la siguiente sección se analizará el tipo de datos de conjunto, que permite escribir una solución elegante para este problema.

## SECCIÓN 8.5 INTRODUCCIÓN AL TIPO DE DATOS DE CONJUNTO

Considérese al siguiente problema: se debe leer y clasificar un carácter como dígito decimal (por ejemplo ‘0’ u ‘8’) o como otro tipo de carácter. Una forma de resolver este problema es emplear una proposición IF que determine si el carácter que se va a clasificar está en la escala de los dígitos decimales (recuérdese que el Pascal requiere que los caracteres que corresponden a los dígitos decimales estén en una sola secuencia contigua). Esta proposición se podría escribir como sigue, suponiendo que el carácter que se va a clasificar está en la variable *uncar*.

```
IF (uncar >= '0') AND (uncar <= '9')
THEN writeln ('El carácter es un dígito.')
ELSE writeln ('El carácter no es un dígito.')
```

Otra estrategia que se puede usar en los programas en Pascal es construir un valor de un tipo de datos llamado *conjunto (set)*. Un conjunto es un grupo de objetos del mismo tipo de datos ordinal. Por ejemplo,

[‘0’..‘9’]

representa al conjunto que contiene la representación de caracteres de todos los dígitos decimales. Este conjunto se puede utilizar para determinar si el carácter (*uncar*) es uno de los caracteres del conjunto. La solución del problema anterior se podría escribir así:

```
IF uncar IN [‘0’..‘9’]
THEN writeln ('El carácter es un dígito.')
ELSE writeln ('El carácter no es un dígito.')
```

La expresión

[‘0’..‘9’]

se llama *constructor de conjunto*, porque produce un valor del tipo de datos de conjunto (*set*), así como ‘X’ produce un valor del tipo de datos de carácter (*char*). Como se verá, los constructores de conjuntos forman conjuntos que contienen un número variable de miembros de acuerdo con los valores ordinales que se incluyen en el constructor.

La expresión

*uncar* IN [‘0’..‘9’]

se cumple si el valor de *uncar* es miembro del conjunto de dígitos “[‘0’..‘9’]” y será falsa en caso contrario. En este caso, el conjunto de dígitos se especifica me-

dianente una pareja de valores de caracteres separados por “..”. Esto quiere decir que la pareja de caracteres, y todos los caracteres intermedios, están en el conjunto. La palabra reservada IN (en) es un operador relacional cuyo resultado es *true* si el valor a su izquierda está en el conjunto que aparece a su derecha.

Los constructores de conjuntos son listas (que podrían estar vacías) de expresiones ordinales, o parejas de expresiones ordinales separadas por “..”, encerradas en paréntesis cuadrados [ y ]. Todas las expresiones ordinales que aparezcan entre los paréntesis cuadrados deben ser del mismo tipo. Si se coloca más de una expresión o pareja de expresiones entre los paréntesis cuadrados, deben estar separadas por medio de comas. El constructor de conjunto

[‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’]

equivale a “[‘0’..‘9’]”.

En seguida se muestran otros ejemplos de constructores de conjuntos. Obsérvese que no es importante el orden en que se escriben los elementos individuales de la lista, pero cuando los elementos son constantes normalmente se escriben en orden ascendente para hacer más legible el programa. Tómese nota también de que es preciso escribir una pareja de expresiones ordinales separadas por “..” de tal manera que la segunda expresión de la pareja no sea menor que la primera.

En estos ejemplos, supóngase que ya se ejecutaron las siguientes proposiciones de asignación:

```
i := 5;
j := 10;
c1 := 'A';
c2 := 'D';
```

<i>constructor de conjunto</i>	<i>miembros del conjunto</i>
['A', 'E', 'I', 'O', 'U']	'A', 'E', 'I', 'O', 'U'
[c1..c2]	'A', 'B', 'C', 'D'
[succ(c1)..pred(c2)]	'B', 'C'
['a', 'c', 'A', 'B']	'a', 'b', 'c', 'A', 'B'
[5..10]	5, 6, 7, 8, 9, 10
[i..j]	5, 6, 7, 8, 9, 10
[i, j]	5, 10
[i..i+3, j]	5, 6, 7, 8, 10
[i+1..j]	6, 7, 8, 9, 10
['0', '1', '5', '7']	'0', '1', '5', '6', '7'
[ ]	Ninguno; éste es el <i>conjunto vacío</i> .

Con base en estos ejemplos se puede verificar que las siguientes expresiones producen el valor booleano indicado.

expresión booleana	valor booleano
'A' IN ['A'..'Z']	true
'A' IN [c1..c2]	true
c1 IN [succ(c1)..c2]	false
3 IN [0..10]	true
(j DIV 2) IN [i..j]	true

Es común emplear expresiones booleanas que usan el operador IN para verificar que el selector de CASE corresponda a uno de los valores de la lista de etiquetas constantes. Como ejemplo, considérese el siguiente segmento de código. Aquí el valor de calif debe ser el de una calificación de letra ('A', 'B', 'C', 'D', o 'F'):

```
IF calif IN ['A'..'D', 'F']
THEN CASE calif OF
    'A','B' : writeln ('Muy bien.');
    'C'      : writeln ('Bien.');
    'D'      : writeln ('Regular.');
    'F'      : writeln ('Vuélvelo a intentar.');
END;
```

También es posible verificar que un valor *no* está en el conjunto especificado si se niega el resultado de la prueba. En este caso es menester emplear paréntesis para garantizar que el operador IN se aplique antes del operador NOT. He aquí un ejemplo de la forma correcta de probar si una calificación no es válida:

```
IF NOT (calif IN ['A'..'D', 'F'])      (* correcto *)
THEN writeln ('La calificación no es válida.')
```

Ahora compárese la solución correcta con una *equivocada*:

```
IF calif NOT IN ['A'..'D', 'F'])      (* incorrecto *)
THEN writeln ('La calificación no es válida.')
```

El siguiente problema es una aplicación de los conjuntos:

### Problema 8.5

*Escribase un procedimiento llamado letra que captura la primera vocal mayúscula o minúscula en los datos de entrada. Este carácter se deberá almacenar en el parámetro variable de carácter uncar.*

El código en Pascal que se muestra en seguida resuelve este problema en forma efectiva al leer repetidamente los caracteres de los datos de entrada hasta que uno cumpla con los requisitos.

```

PROCEDURE letra (VAR uncar: char);
(* Leer hasta que se encuentre una vocal mayúscula o minúscula *)
(* en los datos de entrada y colocar esa vocal en uncar. *)
BEGIN
  REPEAT
    read (uncar)
  UNTIL uncar IN ['a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U']
END.

```

El siguiente programa es otra versión del programa de clasificación de textos, que emplea conjuntos. Obsérvese que no se ha modificado la proposición CASE final.

### Problema 8.6

*Volver a escribir el programa clasetexto de la sección 8.4 usando conjuntos.*

```

PROGRAM clasetexto (* modificado *) (input, output);
(* Programa para clasificar los caracteres de un enunciado como *)
(* vocal, consonante o carácter especial y exhibir el total de      *)
(* caracteres, así como los porcentajes.                         *)
TYPE
  clase = (vocal, consonante, especial);
VAR
  uncar : char;                      (* carácter capturado *)
  total,                                (* número total de caracteres *)
  cuentav,                             (* número de vocales *)
  cuentac,                             (* número de consonantes *)
  cuentae : 0..máxint;     (* número de caracteres especiales *)
  categoría : clase;                  (* tipo de carácter *)
BEGIN
  total := 0;                          (* valor inicial de la cuenta total *)
  cuentav := 0;                        (* valor inicial del contador de vocales *)
  cuentac := 0;                        (* valor inicial, contador de consonantes *)
  cuentae := 0;                        (* valor inicial, contador car. especiales *)
  read (uncar);
  WHILE uncar <> '.' DO
    BEGIN
      (* aumentar contador de caracteres totales *)
      total := total + 1;
      (* determinar clase del carácter usando conjuntos *)
      IF uncar IN ['A', 'E', 'I', 'O', 'U']
      THEN categoría := vocal
      ELSE IF uncar IN [' ', ',', ':', ';']
      THEN categoría := especial
      ELSE categoría := consonante;
      (* aumentar el contador de la categoría apropiada *)
    END
  END.

```

```

CASE categoría OF
    vocal : cuentav := cuentav + 1;
    consonante : cuentac := cuentac + 1;
    especial : cuentae := cuentae + 1;
END; (* del case *)
read (uncar)      (* capturar el siguiente carácter *)
END; (* del while *)
(* exhibir resultados *)
writeln ('El total de caracteres es', total:1, '.');
writeln (cuentav / total * 100 : 6 : 2,
         '% fueron vocales.');
writeln (cuentac / total * 100 : 6 : 2,
         '% fueron consonantes.');
writeln (cuentae / total * 100 : 6 : 2,
         '% fueron caracteres especiales,');
END.

```

---

Los conjuntos se pueden combinar para formar conjuntos nuevos con ayuda de varios operadores. También es posible definir tipos de conjuntos que corresponden a cualquier tipo ordinal, incluso tipos enumerados. Estos tipos se examinarán en un capítulo posterior.

## EJERCICIOS DE LA SECCIÓN 8.5

- 1 Enumérense los miembros de cada conjunto.
  - a) [0, 1, 2, 3]
  - b) [0..10, 13, 15]
  - c) ['A'..'J', 'L'..'Z']
  - d) ['a'..'m', 'A'..'M']
  - e) ['0'..'5', '9']
- 2 Determinése cuáles de los siguientes son constructores válidos.
  - a) [0, 1, '9']
  - b) ['A'..'F', 'G']
  - c) (0..100)
  - d) ['0'..'9', 'A'..'Z']
  - e) [0..9, A..Z]
- 3 Determinése el valor de las siguientes expresiones booleanas.
  - a) 'A' IN ['B'..'S']
  - b) 0 IN [-3..5]
  - c) 7 IN [1..4, 6..8]
  - d) '' IN []
- 4 Determinése el valor de las siguientes expresiones booleanas, suponiendo que  $núm = 3$  y  $uncar = 'B'$ .
  - a) 10 IN [3..3\*núm]
  - b) 5 IN [núm + 2..10]

- c) 'A' IN [uncar..'T']  
 d) 'B' IN ['A'..succ(uncar)]
- 5 Determinese el valor de las siguientes expresiones booleanas, suponiendo que *núm* = 100 y *calif* = 'D'.  
 a) NOT (núm IN [1..200])  
 b) NOT (calif IN ['A'..'F'])  
 c) NOT (calif IN ['F'..'Z'])  
 d) (núm IN [99..199]) AND NOT (calif IN ['A'..'C'])
- 6 Determinese la salida del siguiente programa si se supone que los datos de entrada son R2D2.

```

PROGRAM adivinar (input, output);
VAR
  caract : char;
PROCEDURE inspeccionar (VAR uncar : char);
BEGIN
  REPEAT
    read (uncar)
  UNTIL uncar IN ['0'..'9']
END;
(* Programa principal *)
BEGIN
  inspeccionar (caract);
  writeln (caract);
  inspeccionar (caract);
  writeln (caract)
END.

```

- 7 Escríbase un programa en Pascal que use conjuntos, lea una secuencia de 80 caracteres y exhiba el número de caracteres leídos que no sean dígitos.

## SECCIÓN 8.6 TÉCNICAS DE PRUEBA Y DEPURACIÓN

En esta sección se hablará de los problemas que pueden surgir si no se emplean tipos de datos idénticos o compatibles. Además se analizará un problema de validación de datos de entrada.

### Compatibilidad de tipos

Examínense las siguientes declaraciones:

```

TYPE
  califexam = 0..100;
  letra = 'A'..'F';
VAR
  examen : 0..100;

```

```
semestral,  
final : califexam;  
total : char;  
calif : letra;
```

Obsérvese que *final* y *semestral* son del mismo tipo, es decir, *califexam*. En este caso se dice que las dos variables tienen *tipos idénticos*. Considérese ahora el caso de las variables *final* y *examen*. Al parecer, ambas variables tienen el mismo tipo de subescala, 0..100. No obstante, el Pascal *no* considera que las dos tengan tipos idénticos, ya que se definieron explícitamente dos veces. Más bien, se dice que estas variables tienen *tipos compatibles*. Se dice que dos tipos *T<sub>1</sub>* y *T<sub>2</sub>* son compatibles si se cumple alguna de las siguientes condiciones:

- 1 *T<sub>1</sub>* y *T<sub>2</sub>* son del mismo tipo.
- 2 *T<sub>1</sub>* es una subescala de *T<sub>2</sub>*, o *T<sub>2</sub>* es una subescala de *T<sub>1</sub>*.
- 3 Tanto *T<sub>1</sub>* como *T<sub>2</sub>* son subescalas del mismo tipo huésped.

(Existen unas cuantas reglas adicionales que se refieren a la compatibilidad de otros tipos de datos, las cuales se analizarán más tarde.) Por ejemplo, las variables *total* y *calif* tienen tipos compatibles porque el tipo de la variable *calif* es una subescala del tipo de la variable *total*.

Es posible asociar nombres adicionales a un tipo si se escribe una declaración TYPE de la forma

```
TYPE t1 = t2;
```

en este caso, el tipo *t<sub>1</sub>* es idéntico al tipo *t<sub>2</sub>*, ya que la definición del tipo *t<sub>1</sub>* es la misma que la de *t<sub>2</sub>*. Por ejemplo, a resultas de la declaración

```
TYPE  
    a = integer;  
    b = -máxint..máxint;
```

el tipo *a* es idéntico al tipo *integer*, pero el tipo *b* es solamente compatible con el tipo *integer*.

Es importante darse cuenta de que dos variables pueden tener tipos idénticos si se les declara en nombres de tipo idénticos o si sus nombres aparecen en la misma lista en una declaración de variables. Por ejemplo, en el siguiente código las variables *tempalta* y *tempbaja* tienen tipos idénticos, pero *tempmedia* es de un tipo solamente compatible con el de ellas:

```
TYPE  
    temp = -50..150;  
VAR  
    tempbaja : temp;  
    tempalta : temp;  
    tempmedia : -50..150;
```

Puede decirse que las tres variables del código que se presenta en seguida tienen tipos idénticos, pero ninguna otra variable puede tener un tipo idéntico al de ellas, ya que tendría que haberse declarado en la misma lista.

#### VAR

```
tempa,  
tempb,  
tempm : - 50..150;
```

La distinción entre los tipos idénticos y compatibles es importante cuando se especifican parámetros variables en el encabezado de un procedimiento o función. El tipo del argumento verdadero (que aparece en la invocación) debe tener un tipo *idéntico* al del parámetro variable formal. Por ejemplo, en el encabezado de procedimiento

```
PROCEDURE indicetemp (VAR tempa: temp);
```

el parámetro de variable *tempa* tiene un tipo idéntico al de las variables *tempbaja* y *tempalta* de la declaración anterior, por lo que estas dos invocaciones del procedimiento son válidas:

```
indicetemp (tempbaja);  
indicetemp (tempalta);
```

Las invocaciones del procedimiento que utilicen cualquiera de las otras variables (*tempmedia*, *tempa*, *tempb*, *tempm*) serán incorrectas, ya que los tipos de estas variables no son idénticos al tipo del parámetro variable *tempa*.

En el caso de los parámetros de valor se permite una mayor flexibilidad. Cuando se emplea un parámetro de valor con un procedimiento o función, se puede usar cualquier valor de un tipo compatible como parámetro verdadero, siempre que el valor del parámetro verdadero sea uno de los valores permitidos por el tipo del parámetro formal. Por ejemplo, examíñese el siguiente segmento de programa.

#### TYPE

```
alto = 1500..5000;  
medio = 500..2500;  
bajo = 0..1499;
```

#### VAR

```
sube0 : integer;  
sube1 : medio;  
sube2 : bajo;
```

```
PROCEDURE escala (sube: alto);
```

```
...
```

```
END;
```

```
BEGIN (* Programa principal, u otro procedimiento *)
```

```
sube0 := 4900;  
escala (sube0); (* se permite *)
```

```

    sube1 := 1700;
    escala (sube1);      (* se permite *)
    sube2 := 750;
    escala (sube2);      (* NO se permite *)
    sube1 := 750;
    escala (sube1);      (* NO se permite *)
END

```

En cada una de las invocaciones de procedimientos, el valor del parámetro verdadero está en el conjunto de valores permitidos por el tipo del parámetro formal. Tómese nota de que los tipos *bajo* y *alto* son compatibles, puesto que ambos son subescalas del mismo tipo, pero el valor de una variable del tipo *bajo* no puede usarse nunca como argumento verdadero de un procedimiento que espere un parámetro del tipo *alto*.

La última invocación de procedimiento no permitida en el ejemplo contiene dos tipos que no son *compatibles para asignación*. Dados dos tipos *T1* y *T2*, un valor del tipo *T1* se puede asignar a una variable del tipo *T2* si se cumple alguna de las siguientes condiciones:

- 1 *T1* y *T2* son del mismo tipo.
- 2 *T1* es *integer*, o una subescala de *integer*, y *T2* es *real*.
- 3 *T1* y *T2* son tipos ordinales compatibles (incluso *integer*, *Boolean*, *char*, tipos enumerados y subescalas de ellos), y el valor del tipo *T1* está en el conjunto de valores permitidos por el tipo *T2*.

Por ejemplo, examínense las siguientes declaraciones.

```

TYPE
  escala = 1..10;
VAR
  número : 1..10;
  valor : integer;
  final : escala;

```

Los tipos de las tres variables no son idénticos, pero son compatibles. En la proposición de asignación

*valor* := *número* + *final*;

la expresión “*número* + *final*” es compatible para asignación con el tipo de la variable *valor* porque ambos tipos son compatibles. En la proposición de asignación

*número* := *final* + *valor*;

si el valor de *valor* es 15, no se permite la asignación, ya que el valor que produce la expresión está fuera de la escala de valores permitidos por el tipo de subescala 1..10.

Supóngase que se va a leer un carácter y se espera un dígito ('0' a '9'). ¿Cuál es la mejor forma de validar que el dato de entrada es apropiado? Se examinarán tres métodos: 1) fuerza bruta, 2) ciclos y 3) conjuntos. Supóngase que *uncar* se declara como variable de caracteres, y *bien* es una variable booleana a la que se asignará el valor *true* si el dato de entrada es satisfactorio.

### Método 1: fuerza bruta

```
read (uncar);
bien := (uncar = ('0') OR (uncar = ('1') OR (uncar = ('2') OR
    (uncar = ('3') OR (uncar = ('4') OR (uncar = ('5') OR
        (uncar = ('6') OR (uncar = ('7') OR (uncar = ('8') OR
            (uncar = ('9');
```

Aunque el método funciona, el lector tiene razón si piensa que existe una forma mejor de hacerlo.

### Método 2: ciclos

Supóngase que *dígito* es una variable de caracteres.

```
bien := false;
read (uncar);
FOR dígito := '0' TO '9' DO
    IF dígito = uncar
        THEN bien := true
```

Esto es más conveniente, pero aún puede mejorarse. Trate el lector de determinar por qué no se puede usar la proposición

*bien := dígito = uncar*

en vez de la proposición IF.

### Método 3: conjuntos

```
read (uncar);
bien := uncar IN ['0'. .'9']
```

Esto es mucho mejor. La validación de datos de entrada mediante conjuntos es el mejor de los tres métodos porque es conciso, legible y eficiente.

He aquí algunos recordatorios de Pascal que pueden ser útiles para la prueba y depuración de programas.

**RECORDATORIOS DE PASCAL*****Definiciones de tipo***

Se colocan entre las declaraciones de constantes y de variables.

No se permiten en la lista de parámetros formales de un procedimiento o función.

***Constantes enumeradas***

No se pueden leer o exhibir directamente.

Deben ser identificadores únicos, diferentes de los empleados con cualquier otro propósito. Tienen valores ordinales determinados por el orden en que aparecen en la definición de tipo; el valor ordinal más bajo siempre es cero.

***Tipos de subescala***

El valor inferior siempre debe ser menor que el valor superior o igual a él.

***Ord***

Da como resultado la posición de su argumento en las constantes del mismo tipo de datos que el del argumento.

***Chr***

Da como resultado el carácter que está en la posición especificada por el argumento entero.

***Pred y Succ***

Dan como resultado el elemento de un tipo de datos cuyo valor ordinal es uno más (*succ*) o uno menos (*pred*) que el valor ordinal del argumento.

Producen valores no definidos cuando se aplica *pred* al elemento más pequeño o se aplica *succ* al elemento más grande de un tipo de datos dado.

***Constructores de conjuntos***

Deben encerrarse entre paréntesis cuadrados.

Deben emplear constantes ordinales o variables del mismo tipo.

Pueden incluir varios elementos, subescalas de elementos separados mediante comas, o ningún elemento (conjunto vacío).

***Operador IN***

Da como resultado *true* si el valor a su izquierda es miembro del conjunto a su derecha. En caso contrario resulta *false*.

No puede ir precedido inmediatamente por el operador NOT (en vez de ello, se encierra la expresión IN entre parentésis y se coloca el NOT antes).

***Tipos idénticos***

Son obligatorios para los parámetros variables formales y los parámetros verdaderos correspondientes.

***Tipos compatibles***

Son obligatorios para los parámetros formales de valor y los parámetros verdaderos correspondientes. Además, el valor del parámetro verdadero debe estar en la escala permitida por el tipo del parámetro formal.

En este capítulo se analizaron varios tipos de datos y sus aplicaciones en programas en Pascal. Específicamente, se presentaron los tipos definidos por el usuario, o enumerados, como mecanismo auxiliar para producir programas legibles y de fácil mantenimiento. Pascal permite dar nombres a tipos de datos ordinales propios del programador mediante definiciones de tipo (TYPE). No obstante, los tipos enumerados tienen ciertas desventajas. Las variables de estos tipos no se pueden leer o exhibir en forma directa, se diseñan exclusivamente para usarse en el interior.

Los tipos de datos de subescala permiten restringir la escala de valores que puede representar una variable. Se hace una verificación automática de escala (que en algunos sistemas de Pascal se debe habilitar explícitamente) cada vez que se almacena un valor nuevo en variables de tipos de datos de subescala. Estas características mejoran la legibilidad y el diseño de los programas.

En este capítulo se analizaron las cuatro funciones ordinales estándar, *pred*, *succ*, *ord* y *chr*, las cuales pueden ser muy útiles al manipular datos de tipos ordinales. La función *ord* da como resultado la posición de su argumento en la lista de constantes asociadas al tipo de datos de que se trata y la función *chr* produce el valor de caracteres cuya posición se especifica mediante el argumento. Estas dos funciones constituyen un mecanismo general para manejar el tipo de datos *char*, ya que permiten hacer conversiones entre las representaciones externa e interna (codificada).

Este capítulo presentó también un problema específico (clasificación de textos) y una solución en Pascal que emplea tipos de datos enumerados. La solución se modificó más adelante para emplear el tipo de datos de conjunto. Se presentaron los constructores de conjuntos y el operador IN para probar de manera sencilla si un valor ordinal es miembro de un conjunto arbitrario de valores ordinales. En la sección de técnicas de prueba y depuración se introdujo el concepto de tipos idénticos y compatibles, así como el concepto de compatibilidad para asignación. Se ilustró la importancia de estos conceptos con ejemplos de transferencia de parámetros a procedimientos y funciones.

El siguiente resumen de los conceptos de Pascal descritos en este capítulo puede servir como referencia en el futuro.

## REFERENCIAS DE PASCAL

- 1 Tipo enumerado: tipo de datos ordinal definido por el usuario.

Ejemplo:

```
TYPE color (rojo, azul, verde);  
      textura (suave, lisa, dura, áspera);
```

- 2 Colocación de las definiciones de tipo:

```
PROGRAM nombre (input, output);  
CONST declaraciones:
```

TYPE definiciones; (\* entre CONST y VAR \*)

VAR declaraciones;

...

- 3 Tipo de datos de subescala: escala restringida de valores.  
Ejemplos:

**TYPE**

```
grupo = (primero, segundo, tercero, último, graduado);
gruposup = tercero..último;
```

**VAR**

```
calif : 0..100;
califletra : 'A'..'F';
númpositivo : 1..máxint;
```

- 4 Funciones ordinales estándar:
- 4.1 *Pred(x)*: predecesor del argumento *x*.
  - 4.2 *Succ(x)*: sucesor del argumento *x*.
  - 4.3 *Ord(x)*: posición del argumento *x* en el tipo de datos ordinal.
  - 4.4 *Chr(x)*: valor de carácter cuya posición ordinal en el tipo de datos *char* está dada por el argumento *x*.

- 5 Conjunto: grupo de valores ordinales (elementos). Los valores de conjunto se pueden crear mediante constructores de conjuntos.  
Ejemplos:

[‘A’, ‘B’, ‘C’, ‘X’, ‘Y’, ‘Z’]

[‘B’..‘R’]

[bajo..medio, alto]

[Lunes..Viernes]

El operador IN resulta *true* si el valor a su izquierda aparece en el conjunto a su derecha.

Ejemplos:

‘a’ IN [‘a’, ‘e’, ‘i’, ‘o’, ‘u’]	(* true *)
3 IN [4..10, 25]	(* false *)
NOT (bajo + 4 IN [bajo..bajo + 50])	(* false *)
NOT (2 IN [-5..0])	(* true *)

- 6 Compatibilidad de tipos: tipos idénticos y compatibles.

Ejemplo:

**TYPE**

```
promedio = integer;
calif = 0..100;
```

**VAR**

```
prueba : 0..100;
examen1, examen2 : calif;
```

```
final : integer;  
prom : promedio;
```

### Variables con

- 6.1 Tipos idénticos: *examen1* y *examen2*; *final* y *prom*
- 6.2 Tipos compatibles: *prueba*, *examen1*, *examen2*, *final*, *prom*

## Avance del capítulo 9

En el siguiente capítulo se analizará el primero de los tipos de datos estructurados, el arreglo. Un arreglo es un conjunto de elementos, todos del mismo tipo, a los cuales se hace referencia mediante un solo identificador. Es indispensable usar este tipo cuando un problema requiere mantener muchos valores relacionados entre sí simultáneamente en la memoria. Se presentarán varias aplicaciones del tipo de datos de arreglo.

## Palabras clave del capítulo 8

compatible para asignación	operador IN
constructor de conjunto	tipo de datos de conjunto
definición de tipo (TYPE)	tipo de datos de subescala
función <i>chr</i>	tipo de datos huésped
función <i>ord</i>	tipo definido por el usuario
función <i>pred</i>	tipo enumerado
función <i>succ</i>	tipos compatibles
funciones ordinales estándar	tipos idénticos

## EJERCICIOS DEL CAPÍTULO 8

### ★ EJERCICIOS ESENCIALES

- 1 En la memoria, los valores de un tipo enumerado se almacenan como enteros que representan el valor ordinal de esa constante en el tipo enumerado. Por ejemplo, si se tiene la declaración.

**VAR** cosas : (cosa1, cosa2, cosa3);

y la asignación

**cosas := cosa3**

entonces el valor almacenado realmente en la localidad de memoria destinada a *cosas* es dos. ¿Cómo explica esto el que en Pascal sea imposible exhibir un valor (por ejemplo, *cosa3*) si es de un tipo enumerado?

- 2 ¿Cuál es la diferencia entre un conjunto y un grupo de variables del mismo tipo? Por ejemplo, considérese el conjunto construido mediante

['A', 'D', 'Y']

y las variables

**VAR**

```
car1, car2, car3 : char;
```

con los valores iniciales:

```
car1 := 'A';
car2 := 'D';
car3 := 'Y';
```

**★ ★ EJERCICIOS IMPORTANTES**

- 3 Algunos sistemas de Pascal tienen límites poco convenientes respecto al tamaño de un conjunto.
  - a) Si no pudiera consultar la documentación de su compilador, ¿cómo haría el lector para determinar el tamaño máximo de los conjuntos?
  - b) Dése un ejemplo de limitación de tamaño de los conjuntos que sea poco conveniente.
- 4 ¿Por qué cree el lector que Pascal no permite definir tipos en los encabezados de procedimientos? Es decir, ¿por qué Pascal no permite construcciones como la que sigue?

PROCEDURE esquemas (VAR cuadro : 0..9);

- 5 ¿Existe alguna técnica en Pascal que pueda emplear el lector para especificar el número deseado de dígitos significativos en los números reales? Algunos sistemas de Pascal permiten más de un tipo de representación de los números reales. Determíñese si se puede hacer esto en el sistema propio y, en caso de ser así, el número de dígitos significativos que se permiten en cada tipo.

**PROBLEMAS DEL CAPÍTULO 8 PARA RESOLUCIÓN EN COMPUTADORA****★ PROBLEMAS ESENCIALES**

- 1 Los datos de entrada contienen tres enteros. Úsese un constructor de conjunto a fin de escribir un programa que determine si el primer entero está en la escala especificada por los enteros segundo y tercero. Puede suponerse que el segundo entero es menor que el tercero o igual a él.

Ejemplo de entrada:

5	12	70
31	-10	50
0	0	0

5 no está en la escala 12. .70  
31 está en la escala — 10. .50  
0 está en la escala 0. .0

- 2 Los datos de entrada contienen el nombre  $D$  de un día de la semana y un entero positivo o negativo  $N$  separados mediante un espacio por lo menos en cada línea. Determíñese el día de la semana  $N$  días después del día  $D$ . Utilícese un tipo enumerado con los días de la semana para representar a  $D$ .

Ejemplo de entrada:

Martes 4  
Sábado 3  
Miércoles — 31

Ejemplo de salida:

4 días después del martes es sábado.  
3 días después del sábado es martes.  
31 días antes del miércoles es domingo.

- 3 La primera línea de datos de entrada contiene un entero positivo  $K$ . Las líneas restantes contienen, cada una, un texto terminado, como siempre, por el carácter de fin de línea. Codifíquese el texto y exhibase el mensaje codificado resultante si se sustituye cada carácter del texto por el carácter susceptible de impresión del conjunto de caracteres del sistema propio cuya posición sea  $K$  caracteres después. Por ejemplo, en el conjunto de caracteres ASCII los caracteres alfabéticos son contiguos, por lo que el mensaje “HOLA” se codificará, si  $K$  es igual a tres, como “KROD”. Se deben incluir los caracteres especiales en el sistema de codificación.

Ejemplo de entrada:

3  
HOLA  
Hola

Ejemplo de salida (si se emplea el conjunto de caracteres ASCII):

KROD  
Krod

- 4 Considérese el sistema de codificación que se desarrolló en el problema 3. Prodúzcase un programa que espere  $K$  como el único entero de la primera línea de datos, seguido de líneas de texto codificado. Decodifíquense los mensajes y exhibase el texto normal resultante.

Ejemplo de entrada:

3  
KROD  
Sydfdo

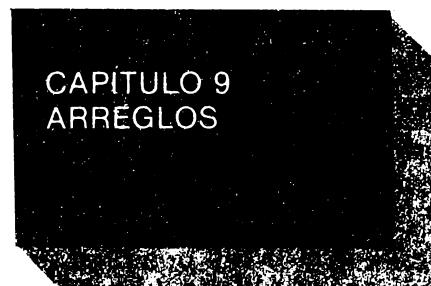
Ejemplo de salida (si se emplea el conjunto de caracteres ASCII):

HOLA  
Pascal

★ ★ PROBLEMA IMPORTANTE

- 5 Escríbase un programa que emplee un tipo de subescala de enteros para simular una limitación del número de dígitos fraccionarios en operaciones aritméticas sobre números reales. (Sugerencia: la multiplicación de un número real por  $10^n$  producirá un entero que, al dividirse entre  $10^n$ , usando aritmética real, dejará solamente  $n$  dígitos fraccionarios.) Úsese este “tipo” nuevo con varies límites para determinar la suma de  $1/2 + 1/3 + 1/4 + \dots + 1/z$ , donde  $z$  es el valor más grande que puede tener un recíproco diferente de cero en el nuevo “tipo”.

# CAPÍTULO 9



## ARREGLOS

## OBJETIVOS

Después de completar este capítulo, el lector deberá ser capaz de:

- Definir y declarar tipos de datos de arreglo
- Reconocer y aplicar las búsquedas lineales y búsquedas binarias
- Reconocer y aplicar un algoritmo de clasificación
- Resolver, probar y depurar problemas que usen arreglos
- De manera opcional, analizar la eficiencia de algunos algoritmos

## PANORAMA GENERAL DEL CAPÍTULO

En este capítulo se analiza uno de los tipos de datos de mayor aplicación en computación, los arreglos. Un *arreglo* es un grupo de elementos o componentes, todos del mismo tipo, a los cuales se hace referencia mediante un índice o subíndice. El grupo completo de elementos recibe un solo nombre.

En la primera sección se presentan los detalles de la definición y declaración de arreglos. Una de las aplicaciones más comunes de los arreglos se encuentra en las búsquedas y clasificaciones. Se presentan dos técnicas de búsqueda, lineal y binaria. Además, se incluye un algoritmo de clasificación y se aplica a un problema. En la sección 9.3 se analizan los arreglos multidimensionales. En la sección 9.4 se resuelven dos problemas mediante el uso de arreglos. La sección de técnicas de prueba y depuración incluye algunos errores comunes que se presentan al usar arreglos, principalmente errores de índice. El capítulo también incluye una sección opcional sobre análisis de eficiencia de algoritmos.

## SECCIÓN 9.1 DECLARACIONES DE ARREGLOS

Los tipos de datos enteros, reales, de caracteres, booleanos, enumerados y de subescala se denominan *tipos de datos simples*. Estos tipos de datos se pueden usar para construir estructuras de datos más complejas llamadas *tipos de datos estructurados*. Pascal incluye las siguientes estructuras de datos integradas: arreglos, registros, archivos y conjuntos, que en lo que resta del curso se examinarán con mayor detalle. Estas estructuras de datos permiten aplicar el Pascal a la resolución de una amplia gama de problemas.

### Arreglo

Un *arreglo* es un tipo de datos estructurado que consta de un grupo de componentes, todos del mismo tipo. Por ejemplo, supóngase que se desea localizar el mayor de un grupo de números. Si se trata de unos cuantos números, el problema puede resolverse mediante proposiciones IF anidadas (véase el Cap. 5). Cada una de las proposiciones IF compararía dos números. Empero, supóngase que se deben ex-

número [1]	(* contiene el primer número, p. ej. 12 *)
número [2]	(* contiene el segundo número, p. ej. 10 *)
número [3]	(* contiene el tercer número, p. ej. 7 *)
número [4]	(* contiene el cuarto número, p. ej. 6 *)
.	.
.	.
.	.
.	.
.	.
.	.
número [1000]	(* contiene el último número, p. ej. 18 *)

minar mil números. Una serie de proposiciones IF sería muy poco práctica, como lo sería declarar las mil variables enteras:

**VAR**

núm1, núm2, núm3, ... núm1000 : integer;

Sería preciso incluir los mil nombres de variable en esta declaración.

Sería preferible agrupar estos números bajo un solo nombre, por ejemplo *número*, y tener acceso a los componentes individuales del grupo mediante *número [1]*, *número [2]* y así sucesivamente hasta *número [1000]* (véase la Fig. 9-1). Este grupo de elementos es un arreglo de enteros.

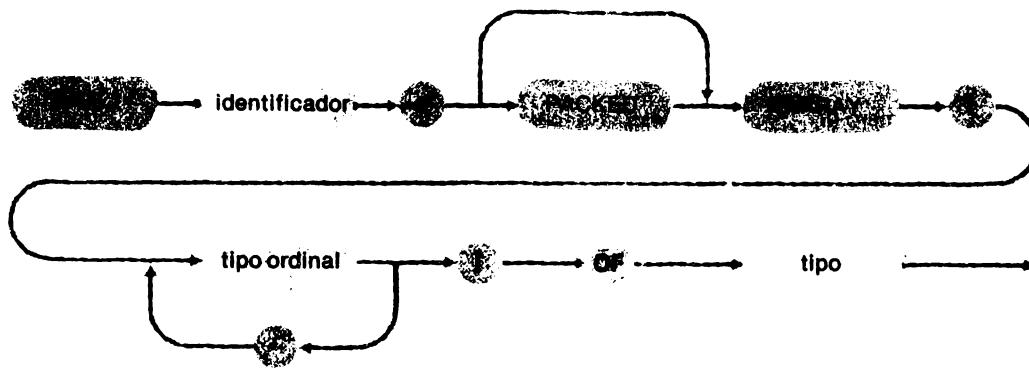
El número que se emplea para distinguir unos componentes de otros se denomina *subíndice*, o *índice*. Por ejemplo, el índice del componente *número [10]* es diez, y el subíndice del componente *número [3]* es tres. El índice sirve para tener acceso a los componentes individuales. El tipo de datos del índice no es forzosamente el mismo que el tipo de los componentes. En este ejemplo, el tipo de datos del índice es 1 .. 1000 y el tipo de datos de los componentes es *integer*.

Con Pascal es posible definir un arreglo de elementos que contiene mil componentes, todos ellos enteros, mediante una definición de tipo (TYPE) como ésta:

**TYPE matriz = ARRAY [1 .. 1000] OF integer;**

La palabra reservada TYPE va seguida de un identificador (*matriz*) que da nombre al tipo de datos estructurado. El identificador del tipo va seguido de un signo de igual y de la palabra reservada ARRAY, la cual va seguida a su vez de la escala del índice encerrada entre paréntesis cuadrados. La escala del índice va seguida de la palabra reservada OF, seguida a su vez del tipo de datos de los componentes. En la figura 9-2 se muestra el diagrama de sintaxis del tipo de arreglo. Los arreglos empacados se analizarán en el capítulo 10.

La siguiente declaración de variable reserva las mil localidades de memoria requeridas para almacenar los números (una vez definido el tipo de datos *matriz* mediante la definición de tipo anterior):



**Figura 9-2** Diagrama de sintaxis de un arreglo.

**VAR** número : matriz;

De hecho, es posible escribir la declaración de variable sin emplear la definición **TYPE**. La declaración anterior se podría escribir así:

**VAR** número : **ARRAY** [1 .. 1000] **OF** integer;

Sin embargo, como se aprendió en el capítulo anterior, esta estrategia no basta cuando se debe pasar la variable de arreglo como parámetro. Es decir, el siguiente encabezado de procedimiento no es correcto:

**PROCEDURE** ejemplo (**VAR** número : **ARRAY** [1 .. 1000] **OF** integer);

La estructura del arreglo se debe definir previamente mediante una declaración **TYPE**. El siguiente encabezado de procedimiento será válido, siempre que se haya escrito la definición **TYPE** de *matriz* previamente:

**PROCEDURE** ejemplo (**VAR** número : matriz);

A continuación se examinarán algunos ejemplos de declaraciones de arreglos.

### Ejemplo 9.1

*Un grupo tiene 50 estudiantes. La declaración de arreglo para almacenar una calificación de letra por cada uno de los estudiantes sería*

**TYPE** arreglolocal = **ARRAY** [1 .. 50] **OF** char;  
**VAR** califletra : arreglolocal;

*o bien*

**VAR** califletra : **ARRAY** [1 .. 50] **OF** char;

Obsérvese que el tipo de datos del índice en este caso es una subescala de los enteros, pero el tipo de datos de los componentes es *char*. La variable *califletra* es una variable de arreglo (o variable con subíndice). Cada uno de los componentes del arreglo se puede emplear de la misma forma que las variables de los tipos de datos simples. Esto implica que pueden aparecer en proposiciones de asignación, proposiciones *read* y *write* y como parámetros verdaderos de procedimientos y funciones. Por ejemplo, las proposiciones de asignación

```
califletra[1] := 'A';
califletra[2] := 'B';
```

asignarán el carácter 'A' al primer componente del arreglo y el carácter 'B' al segundo componente. En el siguiente ejemplo se muestra que la variable de índice puede ser de cualquier tipo ordinal.

### Ejemplo 9.2

*Un programa para contar el número de veces que aparece cada letra mayúscula en una porción de texto podría emplear un arreglo que tenga las letras mayúsculas como índice e integer como tipo de los componentes. El arreglo se podría declarar así:*

```
TYPE cuenta = ARRAY ['A' .. 'Z'] OF integer;
VAR cuentaletra : cuenta;
```

*o bien*

```
VAR cuentaletra : ARRAY ['A' .. 'Z'] OF integer;
```

En este punto el lector podría preguntarse cuándo conviene emplear un índice no entero en un arreglo. Considérese este ejemplo: supóngase que se desea con-

**Figura 9-3** Arreglo con subíndices *char*.

Cuentaletra

['A']	15
['B']	30
['C']	20
['D']	0
.	.
.	.
.	.
.	.
['Z']	0

tar el número de personas que trabajan en determinada compañía cuyo apellido comience con la letra *A*, la letra *B*, etc. Se puede utilizar la declaración de arreglo del ejemplo 9.2 para almacenar estas cuentas. Por ejemplo, la proposición de asignación

```
cuentaletra['T'] := cuentaletra['T'] + 1
```

sumará uno a la cuenta de individuos cuyos nombres comienzan con la letra *T*.

Los arreglos se pueden representar gráficamente como un conjunto de cuadros, un cuadro para cada componente del arreglo. El contenido de cada cuadro representa el contenido de un componente, y el subíndice, o índice, asociado con un componente se escribe junto al cuadro. En la figura 9-3, por ejemplo, puede verse que el número de personas cuyo apellido comienza con *A* es 15, mientras que el número de personas cuyo apellido comienza con *B* es 30.

### Ejemplo 9.3

*Este ejemplo contiene varias declaraciones de arreglo:*

#### TYPE

```
calif = (A, B, C, D, F);
escalat = 0..4;
puntos = ARRAY [1..100] OF escalat;
letra = ARRAY [0..10] OF calif;
escala = ARRAY [A..D] OF real;
```

#### VAR

```
final : calif;
marca : letra;
valor : escala;
```

La siguiente tabla resume las características de los arreglos del ejemplo 9.3.

nombre del arreglo	tipo del arreglo	número de componentes	tipos de datos de componentes
final	1..100	100	escalat (0..4)
marca	0..10	11	calif (A, B, C, D, F)
valor	A..D	4	real

En el ejemplo 9.3 puede verse el uso de varios tipos ordinales para los índices. Cualquier tipo ordinal, incluso subescalas o nombres de tipo completos pueden emplearse para los índices. Como ejemplo final, considérense las declaraciones

#### TYPE

```
calif = (A, B, C, D, F);
cuentacar = ARRAY [char] OF integer;
cuentacal = ARRAY [calif] OF o..maxint;
```

```
frec : cuentacar;
ctacal : cuentacal;
```

¿Cuántos componentes tendrá el arreglo *frec* como resultado de esta declaración? La respuesta depende del sistema de cómputo que se use, pero puede observarse que habrá un componente por cada valor posible del tipo *char*. Por tanto, en muchas máquinas *frec* tendrá 256 elementos. El arreglo *ctacal* tendrá exactamente cinco elementos, ya que hay cinco valores posibles del tipo *calif* (A, B, C, D, F).

En este punto el lector ya debe comprender bien la diferencia entre el tipo de datos que se pueden almacenar en los componentes de un arreglo y el tipo de datos que se emplea para seleccionar (indizar) un arreglo con el fin de localizar un componente determinado.

### Procesamiento de arreglos

Ahora que ya se vio la forma de declarar un arreglo, se pueden examinar algunos ejemplos que manipulan los componentes de un arreglo. Supónganse las siguientes declaraciones:

TYPE

```
matriz = ARRAY [1..1000] OF integer;
letra = ARRAY ['A'..'Z'] OF real;
```

VAR

```
número : matriz;
escala : letra;
índice : 1..1000;
índicecar : 'A'..'Z';
```

Si se usan las construcciones cíclicas es posible procesar los arreglos elemento por elemento. El ciclo FOR que se presenta en seguida leerá mil números y los almacenará en el arreglo llamado *número*, elemento por elemento.

```
FOR índice : = 1 TO 1000 DO
  read (número[índice]);
```

El siguiente ciclo FOR asignará a todos los componentes del arreglo *escala* el valor inicial cero:

```
FOR índicecar : = 'A' TO 'Z' DO
  escala[índicecar] : = 0.0;
```

El siguiente ciclo FOR exhibirá el doble del contenido de todos los elementos del arreglo *número*:

```
FOR índice : = 1 TO 1000 DO
  writeln ('El doble del valor es', 2 * número[índice]);
```

Por último, el siguiente ciclo FOR exhibirá el contenido del arreglo *escala* en orden inverso:

```
FOR índicecar : = 'Z' DOWNTO 'A' DO
    writeln (escala[índicecar]);
```

Ocasionalmente será necesario copiar el contenido de un arreglo en un segundo arreglo con el mismo tipo de componentes. El siguiente problema que se resolverá es un ejemplo de este tipo.

### Problema 9.1

*Escríbase un procedimiento en Pascal que copie el contenido del arreglo puntos, y almacene la copia en el arreglo temp. Supóngase que ambos arreglos son del tipo char y que el tipo índice de cada uno es la subescala -10..10.*

Para resolver este problema es preciso definir globalmente el tipo del arreglo, ya que los arreglos serán parámetros verdaderos del procedimiento:

```
TYPE lista = ARRAY [-10..10] OF char;
```

El procedimiento mismo puede escribirse así:

```
PROCEDURE copiar (VAR : puntos, temp : lista);
(* Copiar el arreglo puntos en el arreglo temp *)
VAR índice : -10..10;
BEGIN
    FOR índice : = -10 TO 10 DO
        temp[índice] : = puntos[índice]
END
```

Obsérvese que se declararon *punto* y *temp* como parámetros formales VAR, a pesar de que *puntos* se usó solamente como parámetro de entrada. La razón de esto es que si se declarara a *puntos* como parámetro de valor, se haría una copia de todo el arreglo para garantizar que el parámetro verdadero de arreglo no se pueda modificar. Esta copia (que se crea) automáticamente en el caso de los parámetros de valor) se copiaría de nuevo (dentro del procedimiento) en el parámetro de variable *temp*. En el caso de un arreglo de 21 elementos este copiado adicional no es necesariamente costoso; no obstante, el copiado de los parámetros de valor puede consumir tiempo y memoria excesivos cuando se emplean parámetros de arreglo grandes y procedimientos de uso frecuente. En general, si el procedimiento llamado no va a modificar un parámetro grande (en términos de requerimientos de memoria), o si la modificación que realiza el procedimiento debe verse reflejada en el parámetro verdadero, dicho parámetro debe ser un parámetro VAR.

Otra solución al problema 9.1 implica una asignación al arreglo completo. Puesto que *temp* y *puntos* tienen exactamente el mismo tipo, se permite escribir

```
temp := puntos
```

lo que hará que cada uno de los componentes de *temp* contenga el mismo valor que el elemento correspondiente de *puntos*. Pero téngase en cuenta que no se permite leer o exhibir un arreglo entero en el Pascal estándar con una proposición parecida a ésta:

### **red (puntos)**

Para leer un arreglo completo debe usarse un ciclo que lea cada uno de los componentes individuales (mediante un índice) del arreglo. Por ejemplo, para leer el arreglo *puntos* normalmente se utilizará el siguiente código, en el que se supone que *índice* es una variable entera.

```
FOR índice := -10 TO 10 DO
    read (puntos[índice])
```

### **Arreglos paralelos**

Muchos problemas requieren el procedimiento simultáneo de más de un arreglo. Por ejemplo, supóngase que un profesor tiene un grupo de 50 estudiantes y asigna calificaciones de letra a las tareas y calificaciones numéricas a los exámenes. El profesor desea escribir un programa en Pascal que exhiba, para cada estudiante, la calificación de letra obtenida en la tarea final y la calificación numérica lograda en el examen final. El profesor almacenó las calificaciones de letra y de número en arreglos separados. Las definiciones de tipo y declaraciones de variable que se usan para los arreglos son:

```
TYPE
    arrletra = ARRAY [1..50] OF 'A'..'F';
    arrnúm = ARRAY [1..50] OF 0..100;
VAR
    califletra : arrletra;
    califnúmero : arrnúm;
```

En este problema *califletra* y *califnúmero* se llaman **arreglos paralelos**. Esto se debe a que cada elemento de *califletra* está apareado con un elemento de *califnúmero*. Se pueden exhibir ambas calificaciones de un estudiante si se emplea el mismo índice con los dos arreglos. El siguiente segmento en Pascal podría emplearse para resolver el problema, suponiendo que se declaró *índice* como variable entera:

```
writeln ('Número de Calificación Calificación');
writeln ('estudiante de letra de número');
writeln ('----- ----- -----');
FOR índice := 1 TO 50 DO
    writeln (índice:4, ':10, califletra[índice], ':12, califnúmero[índice]);
```

El siguiente ejemplo de procesamiento paralelo de arreglos también emplea los arreglos *califletra* y *califnúmero*. En este caso, el profesor desea saber cuántos estudiantes están en cada una de las siguientes categorías:

- 1 Obtuvo calificación numérica menor de 90 y calificación de letra menor que A.
- 2 Obtuvo calificación numérica menor de 90 y calificación de letra de A.
- 3 Obtuvo calificación numérica de 90 o más y calificación de letra de A.
- 4 Obtuvo calificación numérica de 90 o más y calificación de letra menor que A.

Se creará un arreglo adicional llamado *categoría* para almacenar el número de estudiantes en cada categoría. Su declaración es la siguiente:

```
VAR
    categoría : ARRAY [1..4] OF 0..50;
```

La solución a este problema se muestra en seguida. Obsérvese el uso de las proposiciones CASE anidadas para simplificar el problema de agrupar las calificaciones de los estudiantes.

```
FOR índice := 1 TO 4 DO
    categoría[índice] := 0;
FOR índice := 1 TO 50 DO
CASE califnúmero[índice] < 90 OF
    false: CASE califletra[índice] = 'A' OF
        false: categoría[4] := categoría[4] + 1;
        true: categoría[3] := categoría[3] + 1
    END;
    true: CASE califletra[índice] = 'A' OF
        false: categoría[1] := categoría[1] + 1;
        true: categoría[2] := categoría[2] + 1
    END
END
```

El procesamiento de arreglos paralelos se puede llevar a cabo con más de dos arreglos. En el ejemplo previo, el profesor podría haber incluido un arreglo para almacenar los resultados de cada examen, no solamente el final. Una vez más, puede usarse el mismo índice o subíndice para tener acceso a todas las calificaciones de un estudiante.

### Resumen de procesamiento de arreglos

En síntesis, los puntos importantes que conviene recordar en cuanto a la definición y manipulación de los arreglos son:

- Los tipos de arreglo se deben definir mediante una definición de tipo. La forma usual es similar a

```
TYPE nombre = ARRAY [tipo del índice] OF tipo de los componentes;
```

Por ejemplo, para crear un nuevo tipo que represente un conjunto de números reales indizado mediante enteros en la escala de -50 a 50, se usaría la siguiente definición:

- Las variables de un arreglo se pueden declarar mediante una referencia a un tipo de arreglo previamente definido o al escribir explícitamente el tipo del arreglo en la declaración de variable. Por ejemplo, la variable *valor* se puede declarar de manera que tenga 101 elementos reales, indizados con los enteros desde −50 hasta 50 al escribir

**VAR valor : ARRAY [−50..50] OF real;**

o bien

**VAR valor : lista;**

donde *lista* es el tipo que se definió antes.

- El tipo del índice de un arreglo puede ser cualquier tipo ordinal.
- El tipo de datos de los componentes de un arreglo puede ser cualquier tipo.
- El número de valores posibles en el tipo de datos del índice determina el número de componentes individuales del arreglo; existirá un componente para cada valor del tipo de datos del índice.
- Para tener acceso a un componente determinado de un arreglo se emplea la forma

nombre del arreglo [ expresión del tipo del índice ].

Cabe hacer notar que se puede usar como índice una expresión tan compleja como se desee, pero debe producir un valor del tipo que se especificó al declarar el arreglo.

- En los encabezados de procedimientos o funciones se deben declarar parámetros formales de arreglos que tengan un tipo idéntico al de los parámetros verdaderos correspondientes. Por tanto, es necesario dar el nombre al tipo de arreglo y definirlo globalmente con respecto a la declaración de parámetros verdaderos y a la declaración del procedimiento. En muchos casos será apropiado usar parámetros variables para evitar el copiado de arreglos completos que se hace cuando se trata de parámetros de valor.

## EJERCICIOS DE LA SECCIÓN 9.1

- 1 Determíñese el número de componentes que tiene cada uno de los arreglos incluidos en la siguiente declaración.

**TYPE**

```
letra = (A, B, C, D, F);
gama = −3..3;
escala = ARRAY [0.99] OF letra;
matriz = ARRAY [gama] OF char;
línea = ARRAY [letra] OF letra;
```

## VAR

```
arreglox : gama;
arregloy : matriz;
arregloz : linea;
```

- 2 ¿Cuántos números se leen e introducen al arreglo *número*?

## TYPE

```
lista = ARRAY ['0'..'9'] OF integer;
```

## VAR

```
núm, : lista;
índice : '0'..'9';
```

## BEGIN

```
índice := '0';
```

## REPEAT

```
read (núm[índice]);
índice := succ(índice)
```

```
UNTIL (índice = '9')
```

## END

- 3 ¿Cuáles de las siguientes asignaciones son válidas, dada la declaración que se muestra?

## TYPE

```
lista = ARRAY ['A'..'Z'] OF integer;
datos = ARRAY [-3..3] OF char;
```

## VAR

```
lista1, lista2 : lista;
datos1 : datos;
```

- a) lista1['A'] := lista2['B'];
- b) lista1['Z'] := lista2['A'] + 1;
- c) lista1['A'] := datos[0];
- d) lista1[datos[0]] := 0;
- e) lista := lista2;
- f) datos1 := lista;
- g) datos1[lista1['A']] := 'A';

- 4 Supóngase que  $i=3$  y  $j=2$ . Determinese el valor del índice de cada expresión para el arreglo llamado *tabla*.

- a) tabla[i+j]
- b) tabla[7-i+j]
- c) tabla[2\*i+succ(j)]
- d) tabla[i\*j+pred(7)]

- 5 Dadas las declaraciones

## VAR

¿cuál segmento de programa encontrará el valor más pequeño de este arreglo, y almacenará el subíndice del elemento donde está guardado este valor?

- a) FOR prueba := 1 TO 500 DO  
IF (puntos[prueba] < menor)  
THEN menor := puntos [menor];
- b) FOR prueba := 1 TO 500 DO  
IF (puntos[prueba] < menor)  
THEN menor := prueba;
- c) menor := puntos[1];  
FOR prueba := 2 TO 500 DO  
IF (puntos[prueba] < menor)  
THEN menor := puntos[prueba];
- d) índice := 1;  
FOR prueba := 2 TO 500 DO  
IF (puntos[prueba] < puntos[índice])  
THEN índice := prueba;
- e) índice := 1;  
FOR prueba := 2 TO 500 DO  
IF (puntos[prueba] < índice)  
THEN índice := prueba;

6 Considérese el segmento

```
TYPE  
    a = ARRAY [1..4] OF integer;  
VAR  
    x, y : a;  
    k, j : integer;  
BEGIN  
    FOR k := 1 TO 4 DO  
    BEGIN  
        read (x[k]);  
        FOR j := k TO 4 DO  
            read (y[j])  
    END  
END
```

- c) x: 1 6 10 13                            d) x: 1 6 11 13  
                 y: 2 7 12 14                            y: 2 7 11 14  
       e) Demasiados datos de entrada. f) Faltan datos de entrada.  
     El sistema se cae.

7 ¿Cuáles de los siguientes segmentos de programa invertirán un arreglo de caracteres llamado *cadena* que consta de 10 elementos?

- a) FOR *conmuta* := 1 TO 10 DO

```
BEGIN
      temp := cadena[conmuta];
      cadena [11-conmuta] := temp
END
```

- b) FOR *conmuta* := 1 TO 10 DO

```
BEGIN
      temp := cadena[conmuta];
      cadena[conmuta] := cadena[11-conmuta];
      cadena[11-conmuta] := temp
END
```

- c) FOR *conmuta* := 1 TO 5 DO

```
BEGIN
      temp := cadena[conmuta];
      cadena[conmuta] := cadena[11-conmuta];
      cadena[11-conmuta] := temp
END
```

- d) FOR *conmuta* := 1 TO 5 DO

```
BEGIN
      temp := cadena[conmuta];
      cadena[conmuta] := cadena[11-conmuta];
      cadena[conmuta] := temp
END
```

- e) FOR *conmuta* := 1 TO 5 DO

```
BEGIN
      temp := cadena[conmuta];
      cadena[conmuta] := cadena[5-conmuta];
      cadena[5-conmuta] := temp
END
```

8 Escribase una función en Pascal llamada *alfa* que tenga como parámetros un arreglo de caracteres llamado *frase* (con un máximo de 100 caracteres), un entero llamado *largo* (número de caracteres en la frase) y una variable de carácter llamada *letra*. La función *alfa* produce el número de apariciones del carácter *letra* en el arreglo *frase* (inclúyanse todas las definiciones TYPE fuera de la función).

## SECCIÓN 9.2 BÚSQUEDA Y CLASIFICACIÓN

Los problemas que requieren búsquedas en arreglos y la clasificación y ordenamiento de los mismos son muy comunes en las aplicaciones de la computación.

La **búsqueda** de un elemento en un arreglo que satisface una condición especificada es el proceso de determinar si un elemento en particular es o no es componente del arreglo. La **clasificación** de un arreglo es el proceso de reacomodar los componentes del arreglo de manera que se cumpla una relación especificada entre elementos adyacentes del arreglo. Por ejemplo, un problema de clasificación común es organizar los componentes de manera que el segundo componente de cada pareja de componentes adyacentes sea igual que el primer componente o mayor que él. En esta sección se estudiarán ambos temas.

## Búsqueda lineal

Supóngase que una fábrica de piezas de repuesto para automóvil tiene un programa de computadora que emplea un arreglo para mantener los números de parte de las piezas en su inventario. Este arreglo se podría representar gráficamente como se muestra en la figura 9-4. Supóngase además que todos los números de parte son números de cuatro dígitos y el primero es distinto de cero, que existen cien números de parte en el arreglo y que no existe relación alguna entre cada pareja adyacente de números de parte en el arreglo; es decir, el arreglo de números de parte no tiene un orden evidente. Existen 9000 números de parte posibles (números de cuatro dígitos en la escala de 1000 a 9999), pero este almacén en particular tiene únicamente 100 piezas diferentes. ¿Cómo puede determinarse si un número de parte de cuatro dígitos concuerda con alguno de los números de parte que este almacén tiene en existencia?

La solución más sencilla a este problema es comparar el número de cuatro dígitos solicitado con todos los números de parte almacenados en el arreglo, en forma secuencial, hasta que se encuentra el número de parte solicitado o bien se examinan infructuosamente todos los componentes (números de parte) del arreglo. Este tipo de búsqueda se llama **búsqueda lineal** o **secuencial**.

Por ejemplo, supóngase que el arreglo en cuestión tiene un índice de 1..100, y los primeros seis elementos son éstos:

índice	1	2	3	4	5	6
númparte	8567	4612	7714	1362	1002	5117

Se podría preguntar si la pieza de repuesto número 1002 está en existencia en el almacén o si el número 1002 aparece en el arreglo. La búsqueda lineal preguntaría primero si *númparte[1]* es igual a 1002. Puesto que esto no es verdad y hay más componentes en el arreglo, la búsqueda lineal preguntaría después si *númparte[2]* es igual a 1002. Una vez más, esto no es verdad, y todavía no hay componentes en el arreglo, por lo que la búsqueda continúa de manera similar hasta llegar al componente *númparte[5]*. Este componente sí es igual a 1002, por lo que la búsqueda terminará con éxito.

Si se estuviera buscando un número de refacción que no esté en el arreglo, sería preciso examinar los cien componentes del arreglo antes de producir una respuesta negativa. Si se examina un arreglo grande en esta forma, el tiempo requerido

## Númparte

[1]	<b>8567</b>
[2]	<b>4612</b>
[3]	<b>7714</b>
[4]	<b>1362</b>
[5]	<b>1002</b>
[6]	<b>5117</b>
.	....
.	....
.	....
[100]	<b>6888</b>

**Figura 9-4** Arreglo de números de refacción.

para obtener la respuesta a la pregunta será excesivo, pero en el caso de arreglos pequeños la búsqueda lineal es satisfactoria.

En seguida se presenta el algoritmo de la búsqueda lineal. Se emplean dos variables booleanas para controlar el ciclo. La variable *está* será *true* si se localizó el componente deseado y *false* en caso contrario. La variable *listo* tendrá el valor *true* en el momento de completar la búsqueda, ya sea por haber hallado el componente solicitado o porque no haya más componentes que examinar. La variable *listo* tendrá el valor *false* en tanto no se haya localizado el componente deseado y quedan todavía componentes por examinar. Una variable adicional llamada *índice* tendrá el mismo tipo que el índice del arreglo en el que se hace la búsqueda.

*Algoritmo de búsqueda lineal*

Asignar *false* a *listo* y *está* y asignar uno a *índice*.

Mientras no (*listo* o *está*)

(¿se debe seguir buscando?)

Si el valor buscado está en el elemento “*índice*” (sí, ¿ya se halló?)

Entonces asignar *true* a *está* (sí, la búsqueda termina)

Si no, si *índice* < tamaño del arreglo (no, ¿se probaron ya todos?)

Entonces *índice* = *índice* + 1

Si no, asignar *true* a *listo*.

Si el arreglo en el que se va a hacer la búsqueda tiene como declaración de variable

númparte : ARRAY [1..50] OF Integer

entonces la variable *índice* tendrá la declaración de variable

índice : 1..50

Es decir, la variable *índice* tiene el mismo tipo que el índice del arreglo. Así, el código de la búsqueda lineal para localizar un entero de cuatro dígitos llamados *hállame* será

```

listo := false;      (* todavía no se ha terminado *)
está := false;      (* y no se ha encontrado a "hállame" todavía *)
índice := 1;         (* comenzar por el primer elemento del arreglo *)
WHILE NOT (listo OR está) DO
    IF númparte[índice] = hállame
    THEN está := true
    ELSE IF índice <= 50
        THEN índice := índice + 1
    ELSE listo := true

```

Volviendo al almacén de piezas de repuesto para automóvil, se examinará ahora una función completa cuyo fin es localizar un elemento determinado (*númparte*). Supóngase que se hicieron las siguientes definiciones globales:

```

CONST
    tamarr = 100;           (* tamaño del arreglo *)
TYPE
    número = 1000..9999;    (* un número de refacción *)
    lista = ARRAY [1..tamarr] OF número;

```

El resultado de la función *busca* (que se muestra en seguida) será el índice del elemento que se localice o cero si no se encuentra el elemento. Los parámetros de entrada de esta función son:

*númparte* (arreglo de números de pieza (como parámetro VAR))  
*elem* (el número de pieza que se espera encontrar en *númparte*)  
*tamarr* (el número de elementos del arreglo.)

El resultado de la función será el subíndice del elemento en *númparte* si se localizó y cero en caso contrario.

```

FUNCTION busca (tamarr : integer;
                elem : número;
                VAR númparte : lista) : integer;
(* búsqueda lineal del elemento en el arreglo númparte [1..tamarr] *)
(* resultado: índice del elemento en el arreglo o cero si no está *)
VAR
    listo, está : Boolean;
    índice : integer;
BEGIN
    listo := false;      (* todavía no se ha terminado *)
    está := false;      (* y no se ha encontrado a "hállame" todavía *)
    índice := 1;         (*Comenzar por el primer elemento del arreglo *)
    WHILE NOT (listo OR está) DO

```

```

IF númparte[índice] = elem
THEN está := true
ELSE IF índice < tamarr
    THEN índice := índice + 1
    ELSE listo := true;
IF está
THEN busca := índice
ELSE busca := 0
END;

```

Supóngase que se quisiera saber si el número de pieza 1002 está en el arreglo de números de pieza. La siguiente proposición de asignación en el programa principal almacenará el índice del número hallado en la variable *lugar*:

**lugar := busca (tamarr, 1002, númparte)**

Entonces se podría usar el siguiente código para exhibir un mensaje apropiado:

```

IF lugar <> 0
THEN writeln ('Ya se encontró el número de pieza')
ELSE writeln ('La pieza no está en existencia.')

```

¿Qué sucedería si se tratará de usar un ciclo FOR en vez del ciclo WHILE? El siguiente ciclo FOR realiza la tarea, pero seguirá ejecutándose aun después de haberse encontrado el elemento. Esto implica un desperdicio de tiempo, ya que si el elemento está en el arreglo bastará con examinar en promedio la mitad de los elementos (véase el problema 3 al final del capítulo).

```

está := false;
busca := 0;
FOR índice := 1 TO tamarr DO
IF elem != númparte[índice]
THEN BEGIN
    está := true;
    busca := índice
END

```

### Búsqueda binaria

Aunque lo más importante es que los programas sean correctos, también es preciso tener en cuenta la eficiencia de las soluciones. En el ejemplo anterior se supuso que los componentes del arreglo no estaban ordenados, es decir, que no había forma de predecir la relación que existe entre dos componentes adyacentes del arreglo. En cambio, si los componentes del arreglo están ordenados, es posible usar un algoritmo de búsqueda mucho más eficiente para localizar un elemento en los datos: la *búsqueda binaria*. El algoritmo de búsqueda binaria opera en forma similar a una persona que utiliza el directorio telefónico para encontrar el número de teléfono de una determinada persona.

En el directorio telefónico los nombres se encuentran en orden alfabético. Supóngase que se desea localizar a una persona cuyo apellido es Jiménez. No se examinan primero las personas cuyo nombre comienza con *A*, sino que se abre el directorio aproximadamente a la mitad. Si se ve que los nombres en ese lugar comienzan con *L*, se sabrá que hay que seguir buscando, pero solamente en la primera parte del directorio, ya que la *J* está antes de la *L* en el alfabeto. Supóngase que a continuación se abre el directorio cerca de la parte media de la primera mitad, y se ve que los nombres comienzan con *H*. Se sabe entonces que se avanzó demasiado hacia el principio del directorio, y la búsqueda se dirigirá a la parte del directorio entre la *H* y la *L*. En el caso del directorio telefónico se continúa esta búsqueda hasta que se llega a la página donde se debe encontrar Jiménez y entonces (mediante un proceso que equivale a una búsqueda lineal) se localiza (o no se localiza) a Jiménez.

El algoritmo de búsqueda binaria utiliza el mismo principio: comenzar en la mitad del arreglo para localizar el elemento. Si el elemento no está precisamente en la posición media, se repite la búsqueda, concentrándose esta vez ya sea en la primera mitad o en la segunda, según sea el elemento menor y mayor que el valor del punto medio.

Encontrar el punto medio es fácil si los componentes están numerados del uno al *N*; será sencillamente el componente  $N \text{ DIV } 2$ . Cuando se busca, por ejemplo, entre el componente 40 y el 74, es preciso usar una técnica ligeramente diferente. Se desea localizar al elemento que esté en el punto medio entre 40 y 74. Para hacerlo se obtiene el promedio de 40 y 74,  $(40 + 74) \text{ DIV } 2$ , o sea 57. Con esta técnica se aplicará la búsqueda binaria a un ejemplo.

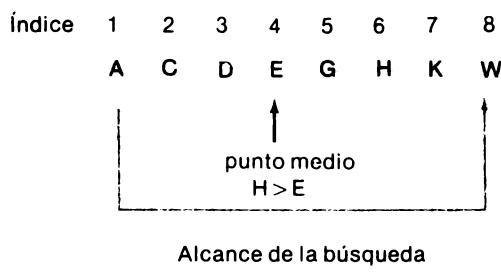
Supóngase que se tiene el siguiente conjunto ordenado de caracteres almacenado en el arreglo *letra* cuyo tamaño es 8:

<i>índice</i>	1	2	3	4	5	6	7	8
<i>letra</i>	'A'	'C'	'D'	'E'	'G'	'H'	'K'	'W'

Ahora se buscará el carácter 'H'. Primero se encontrará el punto medio de todo el arreglo,  $(1 + 8) \text{ DIV } 2$ , o sea 4. La letra que está en esa posición es 'E', que es menor que la letra 'H' que se busca. Por tanto se dirige la búsqueda a la parte del arreglo que contiene letras mayores que 'E', o sea las que se encuentran entre los componentes cinco y ocho, inclusive. Si se calcula una vez más la posición media se obtiene  $(5 + 8) \text{ DIV } 2$ , o sea 6. En esta ocasión se ve que la letra de esa posición concuerda con la letra que se busca, por lo que la búsqueda termina exitosamente (véase la Fig. 9-5).

Se intentará una vez más la búsqueda binaria, pero esta vez con el fin de hallar la letra 'I', que no está presente. La búsqueda de 'I' comenzará exactamente igual que la de la letra 'H'. Cuando se examina el componente seis se encuentra la 'H', no la 'I'. Puesto que 'I' es mayor que 'H', la búsqueda se limita a los componentes del siete al ocho. El "punto medio" de esta escala es el componente siete, la letra 'K', que es mayor que 'I'. No hay más componentes que examinar, por lo que se concluye que la letra 'I' no está en el arreglo.

**Paso 1** Encontrar componente de punto medio y compararlo con H.



**Paso 2** Puesto que  $H > E$ , limitar la búsqueda a la mitad superior.



**Figura 9-5** Ejemplo de búsqueda binaria.

Al principio de la búsqueda no hay manera de saber dónde está el elemento que se busca. Después de la primera comparación se habrá reducido el “área” de búsqueda aproximadamente a la mitad del arreglo original. Después de la segunda comparación sólo queda por examinar la cuarta parte del arreglo original. Este patrón seguirá reduciendo en un 50% el número de componentes que se deben examinar cada vez que se hace una comparación.

Con una búsqueda binaria es posible demostrar que, en el caso de un arreglo de aproximadamente mil componentes, basta con hacer diez comparaciones para determinar si se encuentra en él un elemento determinado, sin importar cuál sea la posición que ocupe en el arreglo. En el caso de la búsqueda lineal es preciso realizar aproximadamente 500 comparaciones, en promedio, para lograr el mismo objetivo. Así, en este caso, la búsqueda binaria es cerca de 50 veces más eficiente (en términos de comparaciones) que la búsqueda lineal. Esta eficiencia mejora todavía más conforme crecen los arreglos. En cambio, en el caso de arreglos pequeños, la ventaja de la búsqueda binaria sobre la búsqueda lineal no es tan impresionante. Por supuesto, la búsqueda binaria requiere que los componentes estén ordenados, requisito que puede provocar problemas importantes en algunas aplicaciones. Más adelante, en esta sección, se examinará con detalle la clasificación de arreglos. Al final del capítulo se hablará de la forma de analizar la eficiencia de los algoritmos.

Ahora se estudiará el seudocódigo de la búsqueda binaria. Se supone que el arreglo que se va a examinar tiene valores de índice entre uno y *ÚLTIMO* y que los componentes están ordenados de manera que el componente *i* tiene un valor inferior o igual al componente *i* + 1, donde *i* está entre uno y *ÚLTIMO* – 1. En este arreglo clasificado se buscará el elemento *elem*.

En el seudocódigo se usan cuatro variables adicionales. Las variables *bajo* y *alto* tendrán los valores de índice de los extremos inferior y superior de la parte del arreglo que se está examinando, y *medio* tendrá el índice del componente con el punto medio de esa porción. La variable *está* es una variable booleana cuyo valor será *true* cuando se haya localizado el componente deseado. He aquí entonces el seudocódigo:

### Algoritmo de búsqueda binaria

Sea *bajo* = 1, *alto* = último y *está* = false.

Mientras (*bajo* <= *alto*) y no está

    Sea *medio* = (*bajo* + *alto*) DIV 2.

(¿prosigue la búsqueda?)

(encontrar punto medio)

    Si *elem* = componente medio,

(¿se halló el elemento?)

        Entonces sea *está* = true,

(sí)

        Si no, si *elem* < componente medio,

(¿está en la mitad inferior?)

            Entonces sea *alto* = *medio* - 1,

(nuevo límite superior)

            Si no, sea *bajo* = *medio* + 1.

(nuevo límite inferior)

    Si *está* es true

        Entonces *elem* está en *medio*,

        Si no, *elem* no está en el arreglo.

A continuación se escribirá una función en Pascal con parámetros similares a los empleados en la búsqueda lineal con el fin de realizar una búsqueda binaria. Se supone que se clasificó previamente el arreglo *númparte*.

```
FUNCTION binaria (tamarr: integer;
                  elem: número;
                  VAR númparte: lista) integer;
(* búsqueda binaria de elem en el arreglo númparte [1..tamarr]. *)
(* El arreglo númparte debe estar clasificado de manera que *)
(* númparte[i]<=númparte[i + 1] si i está en la escala de 1 *)
(* a tamarr - 1. El resultado es el índice de elem en el arreglo *)
(* o cero si no se localizó a elem. *)
VAR
    bajo, alto, medio : integer;
    está : Boolean;
.BEGIN
    bajo := 1;                                (* asignar valor inicial *)
    alto := tamarr;                            (* a las variables *)
    está := false;
    WHILE (bajo <= alto) AND NOT está DO
        BEGIN
            medio := (bajo + alto) DIV 2;          (* ¿hay más
                                                       componentes? *)
            IF elem = númparte[medio]              (* hallar punto medio *)
                THEN está := true                 (* de la escala *)
                ELSE IF elem < númparte[medio]      (* probar si se halló *)
                    THEN alto := medio - 1         (* si, ya se acabó *)
                    ELSE bajo := medio + 1         (* ¿cuál mitad? *)
                                         (* nuevo límite superior *)
                                         (* nuevo límite inferior *)
        END;
        IF NOT está
            THEN binaria := 0                  (* no se encontró elem *)
            ELSE binaria := medio             (* elem está en medio *)
    END;
```

**Clasificación**

A continuación se abordará el problema de la clasificación de arreglos. Como ya se vio, una de las razones por las que es necesario aprender a clasificar arreglos es poderlos preparar para las búsquedas binarias. Producir arreglos clasificados es uno de los problemas que han sido tema de más investigaciones en las ciencias de la computación. Es por ello que existen muchos algoritmos de clasificación diferentes con distintos grados de eficiencia. La elección del algoritmo más apropiado para una situación depende de si existe algún orden en los datos existentes y del número de componentes que se deben ordenar. El análisis de algoritmos de clasificación es un tema que normalmente se reserva para cursos más avanzados en computación, pero se le dedicará una sección opcional en este capítulo.

En esta sección se examinará el algoritmo de *clasificación por selección*. Este algoritmo comienza por localizar el valor más alto en el arreglo que se ha de clasificar e intercambiarlo con el componente que ocupa la última posición del arreglo. Este paso se repite, pero en la parte del arreglo que no incluye al último componente. Esto tendrá el efecto de pasar el segundo valor más alto en el arreglo original a la posición anterior a la última (que contiene el valor más alto). Este proceso se repite hasta que todos los valores se hayan pasado a los componentes apropiados.

Obsérvese que hay que seleccionar el componente más alto del arreglo un total de  $N - 1$  veces, el caso de un arreglo de  $N$  componentes. La razón de esto es que después de ordenar  $N - 1$  valores, el último valor estará forzosamente en la posición correcta.

Considérese el siguiente arreglo de enteros que se debe clasificar:

índice	1	2	3	4	5	6	7	8
arreglo	48	23	56	92	63	90	46	80

En el primer paso hay que seleccionar el valor más alto del arreglo e intercambiarlo con el componente de la octava posición. El número más alto es 92, de manera que se le intercambia con el valor 80 de la última posición. El arreglo se verá ahora así:

índice	1	2	3	4	5	6	7	8
arreglo	48	23	56	80	63	90	46	92

Ahora se repite esta operación básica pero se limita el proceso a la parte del arreglo cuyos valores de índice están entre uno y siete; el valor de la posición ocho ya está correcto. Después de este paso el arreglo tiene el siguiente aspecto:

índice	1	2	3	4	5	6	7	8
arreglo	48	23	56	80	63	46	90	92

El proceso se repite cinco veces más, después de lo cual el arreglo queda clasificado:

índice	1	2	3	4	5	6	7	8
arreglo	23	46	48	56	63	80	90	92

El algoritmo de clasificación por selección depende de la facilidad para localizar el elemento más grande de un arreglo. Supóngase que el arreglo que se debe clasificar contiene  $N$  componentes. Es preciso localizar el número más alto  $N - 1$  veces, cada vez en un arreglo cuyo tamaño es uno menos que en la ocasión anterior. La razón es que después de localizar el valor más alto y colocarlo en la posición correcta ya no es preciso tomarlo en cuenta. El algoritmo de la clasificación por selección, expresado en pseudocódigo, es el siguiente:

*Algoritmo de clasificación por selección*

Para  $j = N$  hasta 2 hacer lo siguiente

Encontrar el valor máximo en las posiciones 1 a  $j$ .

Intercambiar el valor máximo con el de la posición  $j$ .

Para escribir el código en Pascal que realiza una clasificación por selección, se escribe primero una función llamada *hallamáx* cuyo resultado es el índice del componente más alto del arreglo (no el valor del componente más alto). Se supondrá en este ejemplo que el arreglo tiene 500 elementos y que el tipo de los elementos es *integer*:

```
CONST
  tamarr = 500;
TYPE
  lista = ARRAY [1..tamarr] OF integer;
```

La función debe recibir el nombre del arreglo (llamado aquí *tabla*) y el índice de la última posición (*último*) como parámetros.

```
FUNCTION hallamáx (último: integer; VAR tabla: lista): integer;
(* hallar el índice del elemento más grande en tabla[1..último] *)
VAR
  indmáx, índice : 1..tamarr;
BEGIN
  indmáx := 1;
  FOR índice := 2 TO último DO
    IF tabla[índice]>tabla [indmáx]
```

```

THEN indmáx := índice;
hallamáx := indmáx
END;
```

En seguida se escribirá el procedimiento *clasif*. Se necesitan dos parámetros: el tamaño del arreglo y el arreglo mismo. Se supondrá que se definió la función *hallamáx* antes de *clasif*.

```

PROCEDURE clasif (tamarr: integer; VAR tabla: lista);
(* Clasificar por selección los elementos del arreglo tabla[1..tamarr] *)
VAR
    temp, j, mayor: integer;
BEGIN
    FOR j := tamarr DOWNTO 2 DO
    BEGIN
        (* hallar el elemento más grande en 1..j *)
        mayor := hallamáx (j, tabla);
        (* intercambiárselo con tabla[j] *)
        temp := tabla[mayor];
        tabla[mayor] := tabla[j];
        tabla[j] := temp
    END
END:
```

Ahora se juntarán todas estas piezas para escribir un programa completo en Pascal.

### Problema 9.2

*Escribáse un programa completo en Pascal que lea un entero N (no mayor de 500) y N enteros adicionales. Estos N enteros se deben clasificar y después exhibir.*

Ésta es la solución completa al problema en Pascal.

---

```

PROGRAM selección (input, output);
(* Leer N, no mayor de 500, y N enteros. *)
(* Clasificar y exhibir los N enteros.      *)
CONST
    tamarr = 500;
TYPE
    lista = ARRAY [1..tamarr] OF integer;
VAR
    i, núm : 1..tamarr;
    tabla : lista;
FUNCTION hallamáx (último : integer; VAR tabla : lista) : integer;
(* hallar el índice del elemento más grande en tabla[1..último] *)
VAR
    indmáx, índice : 1..tamarr;
```

```

BEGIN
    indmáx := 1;
    FOR índice := 2 TO último DO
        IF tabla[índice] > tabla[indmáx]
            THEN indmáx := índice;
    hallamáx := indmáx
END:
PROCEDURE clasif (tamarr : integer; VAR tabla : lista);
(* clasificar por selección los elementos del arreglo tabla [1..tamarr] *)
VAR
    temp, j, mayor: integer;
BEGIN
    FOR j := tamarr DOWNTO 2 DO
        BEGIN
            (* hallar el elemento más grande en 1..j *)
            mayor := hallamáx (j, tabla);
            (* intercambiárselo con tabla [j] *)
            temp := tabla[mayor];
            tabla[mayor] := tabla[j];
            tabla[j] := temp
        END
    END:
(* Programa principal *)
BEGIN
    writeln ('Clasificación de enteros por selección');
    (* obtener y validar el número de valores que se va a clasificar *)
    REPEAT
        write ('¿Cuántos números (', tamarr, ' o menor)? ');
        readln (núm)
        UNTIL ((núm <= tamarr) AND (núm > 0));
    (* obtener números que se han de clasificar *)
    writeln ('Escriba los números que se deben clasificar.');
    FOR i := 1 TO núm DO
        read (tabla[i]);
    (* invocar la clasificación para ordenar el arreglo *)
    clasif (núm, tabla);
    (* exhibir números clasificados *)
    writeln ('Los números clasificados son: ');
    FOR i := 1 TO núm DO
        writeln (tabla[i], ' ')
END.

```

*Programa selección*

Clasificación de enteros por selección  
 ¿Cuántos números (500 o menos)? 8  
 Escriba los números que se van a clasificar.  
 48 23 56 92 63 90 46 80

Los números clasificados son:

23  
46  
48  
56  
63  
80  
90  
92

---

*Programa selección: ejemplo de ejecución*

## EJERCICIOS DE LA SECCIÓN 9.2

- 1 Considérese la siguiente lista de números:

8 4 3 9 5 12 6

- a) ¿Cuántas comparaciones se requieren para localizar el valor 12 al realizar un búsqueda lineal?
  - b) Clasifíquense los números y determíñese el número de comparaciones requerido para localizar el valor 12 mediante una búsqueda binaria.
- 2 Supóngase que se aplica la clasificación por selección a un arreglo que ya está clasificado en orden ascendente, como el siguiente:

1 2 3 4 5 6

y a un arreglo que está clasificado en orden descendente, como:

6 5 4 3 2 1

Compárese la eficiencia de la clasificación por selección de estos dos arreglos.

- 3 Escríbase un programa en Pascal llamado *fusión* cuyos datos de entrada sean desarreglados aleatorios de enteros de tamaño  $m$  y  $n$ , respectivamente. El programa debe ejecutar los siguientes pasos (se puede emplear la selección por clasificación de este capítulo):
- 1) Clasificar el primer arreglo de tamaño  $m$ .
  - 2) Clasificar el segundo arreglo de tamaño  $n$ .
  - 3) Copiar el primer arreglo a un nuevo arreglo largo de tamaño  $m + n$ .
  - 4) Agregar el segundo arreglo al primero en el nuevo arreglo.
  - 5) Clasificar el nuevo arreglo.
- 4 Escríbase una función similar a *hallamáx* en este capítulo que encuentre el valor más pequeño en vez del más grande.

5. Escríbase un procedimiento de clasificación por selección que clasifique un arreglo de enteros en orden descendente mediante la localización del valor más pequeño en cada iteración y el intercambio con el componente apropiado.
- 

### SECCIÓN 9.3 ARREGLOS MULTIDIMENSIONALES

Los arreglos que se han estudiado hasta ahora son de una sola dimensión, o arreglos lineales, ya que es posible localizar cualquier componente del arreglo mediante la especificación de un solo índice. Sin embargo, existen muchos problemas cuyos datos se representan mejor en un formato multidimensional. Por ejemplo, algunos problemas de procesamiento paralelo de arreglos se pueden considerar como problemas de arreglos multidimensionales. Una tabla con dos o más columnas, por ejemplo, se puede modelar sobre la base de un arreglo bidimensional, a la vez que un conjunto de estas tablas se modelaría como un arreglo tridimensional.

Considérese la siguiente tabla de distancias en millas entre ciudades seleccionadas de Estados Unidos.

de/a	Nueva York	Los Ángeles	Chicago	Houston	Miami
Nueva York	0	2913	841	1675	1350
Los Ángeles	2913	0	2175	1566	2817
Chicago	841	2175	0	1110	1388
Houston	1675	1566	1110	0	1300
Miami	1350	2817	1388	1300	0

Al hablar de tablas de este tipo es frecuente emplear los términos renglón y columna. Un *renglón* es un conjunto horizontal de datos; por ejemplo, las distancias de Nueva York a las diferentes ciudades representan un renglón en la tabla anterior. Una *columna* es un conjunto vertical de datos; las distancias a Nueva York desde las diferentes ciudades son un ejemplo de columna. La tabla completa que se mostró contiene cinco renglones y cinco columnas de distancias. Se puede localizar la distancia entre dos ciudades si se emplean dos valores índice: el nombre de la ciudad de origen (renglón) y el nombre de la ciudad de destino (columna). El primer índice selecciona un determinado renglón de la tabla y el segundo selecciona una columna específica. El elemento seleccionado es el que se encuentra en la intersección del renglón y la columna seleccionadas por los valores índice.

Una forma de imaginar los arreglos multidimensionales es como arreglo de arreglos. Esta idea al parecer extraña, es en realidad bastante lógica. Se puede tratar la tabla de distancias como cinco arreglos, cada uno de los cuales tiene cinco elementos. Por ejemplo, el primer arreglo podría incluir las distancias de Nueva York a cada una de las cinco ciudades y el segundo arreglo podría incluir las distancias desde Los Angeles. Cada uno de estos arreglos no es más que un componente del arreglo más grande. En seguida se estudiará la definición de estos arreglos.

En primer término, la distancia de cualquier ciudad a las cinco ciudades arriba mencionadas se representa mediante cinco enteros. Se usan los nombres de las ciudades como tipo enumerado y este tipo se emplea como tipo índice del tipo de arreglo *renglón*:

**TYPE**

```
ciudad = (NuevaYork, LosÁngeles, Chicago, Houston, Miami);
renglón = ARRAY [ciudad] OF integer;
```

Ahora se pensará en una variable que pueda representar un solo renglón, por ejemplo las distancias entre Houston y las demás ciudades. Este renglón, por sí solo, se declararía como variable mediante la declaración.

**VAR**

```
deHouston : renglón;
```

"

Si se colocaran los valores apropiados en los elementos de este renglón, se podría determinar la distancia de Houston a Nueva York si se hace referencia a “de Houston [Nueva York]”.

Pero para representar la tabla de una manera uniforme, lo que realmente hace falta es un conjunto de estos renglones, uno por cada ciudad de origen. Para declarar un conjunto así sería preciso emplear un tipo que tenga un renglón para cada ciudad:

**TYPE**

```
distancia=ARRAY [ciudad] OF renglón;
```

Por supuesto, una variable de este tipo, que representa a toda la tabla se declararía de la manera usual:

**VAR**

```
millas : distancia;
```

El tipo *distancia* en esta declaración se podría escribir en forma explícita:

**VAR**

```
millas : ARRAY [ciudad] OF ARRAY [ciudad] OF integer;
```

Aunque conviene que el lector siga considerando los arreglos bidimensionales como arreglo de arreglos, también puede escribir el tipo en una forma abreviada. Lo esencial es listar, entre los paréntesis cuadrados, los dos tipos índice empleados. Es decir, se escribe el tipo índice que selecciona el renglón, una coma y el tipo índice que selecciona la columna. Así pues, las distancias se podrían declarar de esta manera:

**VAR**

```
millas : ARRAY [ciudad, ciudad] OF integer;
```

```

TYPE           distancia : ARRAY [ciudad, ciudad] OF integer;
VAR            millas : distancia;

```

La distancia de Nueva York a Houston se localizaría en *millas [Nueva York, Houston]*.

He aquí otro ejemplo en el que son útiles los arreglos bidimensionales. Supóngase que se desea almacenar el siguiente conjunto de nombres:

```

Cardoso
Jiménez
Wileman
Torres
Konvalina
Tedesco

```

En esencia, ésta es una tabla de caracteres en la que cada renglón contiene un nombre completo y cada columna contiene el primer carácter, el segundo, y así sucesivamente, de todos los nombres. Si se supone que ningún nombre tiene más de 15 caracteres (el número de columnas) y que solamente habrá seis nombres, se puede escribir la declaración de tipo apropiada:

```

TYPE
  unombre = ARRAY [1..15] OF char;
  nombres = ARRAY [1..6] OF unombre;

```

o bien

```

TYPE
  nombres = ARRAY [1..6,1..15] OF char;

```

Ahora se declarará una variable de tipo *nombres* y se verá la forma de hacer referencia a sus elementos.

```

VAR gente : nombres;

```

asignaría espacio para seis nombres de quince caracteres cada uno. Se hará referencia al primer carácter del sexto nombre mediante *gente[6][1]*, o bien *gente[6, 1]*. En la primera forma se emplearon de manera explícita dos subíndices (ya que, de hecho, se hace referencia a dos arreglos). El primer subíndice selecciona el renglón que contiene al nombre y el segundo selecciona el carácter específico del nombre en cuestión. La segunda forma es similar a la declaración abreviada; se escriben los dos valores índice separados por una coma en vez de los dos subíndices separados. Los arreglos de caracteres, de una o dos dimensiones, se estudian con mayor detalle en el capítulo 10.

El número de elementos de un arreglo multidimensional es igual al producto del número de valores índice de cada dimensión. La tabla de distancias tiene el mismo tipo índice de renglón y de columna (**ciudad**) y, como éste tiene cinco valores, existirán 25 valores enteros en el arreglo **distancia**, independientemente de la declaración que se use. El arreglo **gente** requerirá 90 caracteres, ya que el primer índice tiene seis valores posibles y el segundo quince. Nótese que, en el caso del arreglo de caracteres, los caracteres que no forman realmente parte de un nombre sí están presentes en el arreglo y son normalmente espacios en blanco.

He aquí algunas declaraciones adicionales de tipo que representan arreglos bidimensionales:

#### TYPE

```
nombre = ARRAY [1..20, 1..15] OF char;
datos = ARRAY ['0'..'9', 1..10] OF integer;
tabla = ARRAY [-10..10, 1..15] OF char;
```

Obsérvese que no es obligatorio que los tipos índice sean iguales para las dos dimensiones de un arreglo bidimensional.

Considérense las siguientes declaraciones y código:

```
VAR núm : ARRAY [1..3, 1..5] OF integer;
FOR i := 1 TO 3 DO      (* por cada renglón *)
    FOR j := 1 TO 5 DO (* por cada columna *)
        núm[i,j] := i;
```

Después de la ejecución de este ciclo, el arreglo tendrá los siguientes valores:

		columna				
		1	2	3	4	5
hilera	1	1	1	1	1	1
	2	2	2	2	2	2
	3	3	3	3	3	3

Obsérvese que se requieren dos índices para procesar arreglos bidimensionales, uno para el renglón y otro para la columna. Los índices pueden ser expresiones o constantes. Por ejemplo, el segmento en Pascal

```
i := 1;
j := 2;
writeln (núm[i + j, j - i])
```

exhibirá el valor que se encuentre en *núm[3,1]*, o sea 3.

#### Procesamiento de arreglos bidimensionales

Si se hace la siguiente declaración

**TYPE**

**tabla** = ARRAY [1..5, 1..1] OF integer;

**VAR**

**valor** : **tabla**;

se podrán capturar valores para introducirlos al arreglo *valor* renglón por renglón mediante el siguiente segmento en Pascal (*renglón* y *columna* se declaran como tipos enteros apropiados):

```
FOR renglón := 1 TO 5 DO
  FOR columna := 1 TO 10 DO
    read (valor [renglón, columna]);
```

Supóngase que se modifica la declaración de manera que el número de renglones sea igual al número de columnas:

```
TYPE
  tabla = ARRAY [1..5, 1..5] OF integer;
VAR
  valor : tabla;
```

Se puede escribir directamente un segmento en Pascal que exhiba los elementos diagonales del arreglo. Los componentes de la diagonal tienen las siguientes posiciones; [1,1] [2,2] [3,3] [4,4] [5,5]. En este caso basta con usar una sola variable índice, ya que la posición en el renglón será igual a la posición en la columna para cada elemento diagonal. He aquí el código, si se supone que *índice* es una variable entera:

```
FOR índice := 1 TO 5 DO
  read (valor[índice, índice]);
```

Supóngase ahora que se desea exhibir la otra diagonal del arreglo, es decir, que se desea exhibir los elementos en las posiciones [5,1] [4,2] [3,3] [2,4] y [1,5]. Para lograrlo, es preciso encontrar la relación que existe entre los números de renglón y de columna. Esta relación deberá ser invariante durante toda la ejecución del ciclo que exhiba los elementos diagonales. Basta pensar un poco para deducir la relación correcta en este caso: el número de renglón más el número de columna siempre es seis. Con base en este hecho, es posible escribir el código para exhibir la diagonal:

```
FOR columna := 1 TO 5 DO
  writeln (valor[6-columna, columna]);
```

Encontrar la relación entre el renglón y la columna en problemas como éste es importante, porque la única otra solución al problema es escribir proposiciones que hagan referencia explícita a los elementos deseados en el orden requerido. Así, para exhibir la otra diagonal, sería menester escribir

```
writeln (valor[5,1];
writeln (valor[4,2];
writeln (valor[3,3];
writeln (valor[2,4];
writeln (valor[1,5];
```

y debe ser obvio que esta solución no puede ser aceptable si el arreglo cuenta con muchos elementos.

### Arreglos de tres o más dimensiones

Es posible emplear arreglos de más de dos dimensiones si son apropiados para el problema que se quiere resolver. Por ejemplo, puede pensarse que un arreglo tridimensional es un arreglo de arreglos de arreglos. En las declaraciones

```
TYPE
    cuadro = ARRAY [1..5, 1..10, 1..5] OF integer;
VAR
    puntos : cuadro;
```

se reservará un total de cinco por diez por cinco enteros, o sea 250. Para tener acceso a un elemento del arreglo *puntos* es preciso usar tres índices, uno para cada dimensión. Por ejemplo, el siguiente código calcula la suma de todos los 250 elementos del arreglo. Supóngase que *i*, *j*, *k* y *suma* se declaran como variables enteras.

```
suma := 0;
FOR i := 1 TO 5 DO
    FOR j := 1 TO 10 DO
        FOR k := 1 TO 5 DO
            suma := suma + puntos[i,j,k];
```

El método de declarar y manejar arreglos multidimensionales se puede ampliar a cualquier número de dimensiones aunque en programas prácticos no es común encontrar arreglos de más de tres dimensiones.

## EJERCICIOS DE LA SECCIÓN 9.3

- 1 Examíñese las siguientes declaraciones de arreglo. ¿Cuántos elementos están presentes en este arreglo?

```
TYPE
    índicereng = -5..5;
    índicecol = 0..10;
    tiporenglón = ARRAY [índicereng] OF integer;
    tipomatriz = ARRAY [índicecol] OF tiporenglón;
```

---

matriz : tipomatriz;

- 2 ¿Cuáles de las siguientes declaraciones *no* declaran a *mitabla* con 100 renglones y 200 columnas?

a) TYPE

porción = ARRAY [1..100] OF char;  
tabla = ARRAY [1..200] OF porción;

VAR

mitabla : tabla;

b) TYPE

porción = ARRAY [1..200] OF char;  
tabla = ARRAY [1..100] OF porción;

VAR

mitabla : tabla;

c) TYPE

tabla = ARRAY [1..100] OF  
ARRAY [1..200] OF char;

VAR

mitabla : tabla;

d) TYPE

tabla = ARRAY [1..100, 1..200] OF char;

VAR

mitabla : tabla;

e) TYPE

elemento = char;  
renglón = 1..100;  
columna = 1..200;  
tabla = ARRAY [renglón, columna] OF elemento;

VAR

mitabla : tabla;

- 3 Determíñese qué es lo que se exhibe al ejecutarse el siguiente programa, si se considera que se hizo la declaración

VAR matriz : ARRAY [1..5, 1..3] OF integer;

FOR i := 1 TO 5 DO

FOR j := 1 TO 3 DO

  matriz[i,j] := i + j;

j := 1;

FOR i := 1 TO 3 DO

  writeln (matriz[j + i,4-i]);

- 4 Examíñese la siguiente declaración:

TYPE

tabla = ARRAY ['0'..'9', 1..9] OF Boolean;

## VAR

```
matriz : tabla;
```

- a) ¿Cuántos componentes contiene esta tabla?  
 ¿Cuáles de las siguientes proposiciones de asignación son válidas?
- `matriz[0,1] := true;`
  - `matriz['9',9] := 0;`
  - `matriz['0',1] := matriz ['9',0] AND matriz ['7',7];`
  - `matriz['1',1] := '1' < 1;`
- 5 Escríbase un procedimiento en Pascal llamado *cambio* que tenga los parámetros *matrizr* (arreglo bidimensional entero de diez renglones y diez columnas) y dos variables enteras *m* y *n*. El procedimiento *cambio* intercambia los renglones *m* y *n* de *matrizr*. Inclúyanse las definiciones de tipo necesarias.
- 6 Escríbase un programa en Pascal llamado *trueque* que haga lo siguiente:
- Leer valores para los enteros *índice1*, *índice2* e *índice3* que serán los valores de índice de un arreglo booleano tridimensional llamado *lógica* (de tipo “ARRAY [1..10, 1..20, 1..30] OF Boolean”);
  - Leer los elementos del arreglo *lógica*; un cero en los datos de entrada representará *false* y un uno representará *true*.
  - Invocar un procedimiento llamado *flipflop* que cambie todos los *true* a *false* y viceversa, en cada uno de los componentes del arreglo *lógica*.

## SECCIÓN 9.4 RESOLUCIÓN DE PROBLEMAS MEDIANTE ARREGLOS

En esta sección se resolverán dos problemas con el uso del material presentado en este capítulo. El primer problema se formuló en el capítulo 1 y se desarrolló un algoritmo. Como el lector ya sabe clasificar arreglos, es posible escribir el programa en Pascal.

### Problema 9.3

*La gerencia de Apex Electronics Company quiere determinar la mediana de los salarios anuales de sus empleados. Escríbase un programa en Pascal que exhiba el número de empleados y la mediana de sus salarios, dados el número de empleados y sus salarios como datos de entrada.*

Recuérdese el algoritmo desarrollado en el capítulo 1:

- Capturar el número de salarios *N* y los salarios de los empleados.
- Clasificar los salarios.
- Calcular la mediana de los salarios como sigue: si *N* es non, la mediana es el salario del punto medio. En caso contrario la mediana será el promedio de los dos salarios del punto medio.
- Exhibir *N* y la mediana de los salarios.

Se supondrá que hay un máximo de 500 empleados en la compañía. El problema se resuelve mediante el siguiente programa en Pascal llamado *estadísticas*. El programa emplea la función *hallamáx* y el procedimiento *clasif* que ya se vieron en este capítulo. El número de empleados se almacena en la variable *núm*.

---

```

PROGRAM estadísticas (input, output);
(* Calcular la mediana de 500 salarios o menos, dados el *)
(* número y valor de los salarios como datos de entrada. *)
CONST
    tamarr = 500;
TYPE
    lista = ARRAY [1..tamarr] OF integer;
VAR
    i, núm : 1..tamarr;
    salario : lista;
    mediana : integer;
FUNCTION hallamáx (último : integer; VAR tabla : lista) : integer;
(* hallar el índice del elemento más grande en tabla[1..último] *)
    VAR
        indmáx, índice : 1..tamarr;
BEGIN
    indmáx := 1;
    FOR índice := 2 TO último DO
        IF tabla[índice]>tabla[indmáx]
        THEN indmáx := índice;
    hallamáx := indmáx
END;
PROCEDURE clasif (tamarr : integer; VAR tabla : lista);
(* clasificar por selección los elementos del arreglo tabla[1..tamarr] *)
    VAR
        temp, j, mayor: integer;
BEGIN
    FOR j := tamarr DOWNTO 2 DO
    BEGIN
        (* hallar el elemento más grande en 1..j *)
        mayor := hallamáx (j, tabla);
        (* intercambiarlo con tabla[j] *)
        temp := tabla[mayor];
        tabla[mayor] := tabla[j];
        tabla[j] := temp
    END
END;
(* Programa principal *)
BEGIN
    writeln ('Cálculo de mediana de salarios');
    (* capturar y validar el número de empleados *)

```

**REPEAT**

```

        write ('¿Cuántos empleados (', tamarr, ', o menos)?');
        readln (núm)
        UNTIL ((núm <= tamarr) AND (núm >0));
(*capturar salarios de empleados *)
        writeln ('Escriba los salarios.');
        FOR i := 1 TO núm DO
            read (salario[i]);
(* invocar clasif para ordenar los salarios *)
        clasif (núm, salario);
(* determinar mediana de los salarios *)
        IF odd(núm)
        THEN mediana := salario[(núm + 1) DIV 2]
        ELSE mediana := (salario[núm DIV 2]
            + salario[núm DIV 2 + 1]) DIV 2;
(* exhibir resultados *)
        writeln ('Número de salarios: ', núm:1);
        writeln ('Mediana de los salarios: ', mediana:1)
END.
```

---

*Programa estadísticas*

-----  
 ¿Cuántos empleados (500 o menos)? 5  
 Escriba los salarios:  
**20000 30000 25000 60000 40000**  
 Número de salarios: 5  
 Mediana de los salarios: 30000  
 -----

*Programa estadísticas: ejemplo de ejecución*

En el siguiente problema se usa un arreglo bidimensional para resolver un problema de juego.

**Problema 9.4**

*Escríbase un procedimiento en Pascal llamado ganador que determine si un juego de “gato” completo tiene ganador y, de ser así, identifique al ganador y el tipo de triunfo (renglón, columna o diagonal). Esto es determinese si un arreglo de tres renglones y tres columnas contiene tres X o tres O en cualquier renglón, columna o diagonal y, en caso afirmativo, cuál letra (X u O) aparece tres veces y de qué manera.*

En la figura 9-6 aparecen ejemplos de juegos de “gato” completos. Existen tres símbolos que pueden aparecer en el juego: X, O y espacio en blanco. La solución a este problema puede reducirse a la resolución de los siguientes subproblemas:

x o o	x o x	x o x	x	o o o
x o x	x o	x o x	x o	x x o
o x o	o x	o x o	x o	x x

Ganador O (diagonal) Ganador X (diagonal) No hay ganador Ganador X (columna) Ganador O (renglón)

Figura 9-6 Juegos de “gato”.

SUBPROBLEMA 1: Buscar tres *X* o tres *O* en cada uno de los renglones.

SUBPROBLEMA 2: Buscar tres *X* o tres *O* en cada una de las columnas.

SUBPROBLEMA 3: Buscar tres *X* o tres *O* en cada una de las diagonales.

Todos los subproblemas se pueden resolver mediante la proposición IF apropiada para verificar si existe un ganador. En seguida se presentan los detalles de la solución en Pascal.

Las definiciones de tipo que se van a necesitar son: la de los diferentes símbolos que pueden aparecer (tipo *marca*), los distintos lugares donde pueden aparecer tres símbolos en fila (tipo *lugar*) y el tablero de juego (tipo *tablero*). He aquí la definición TYPE en Pascal requerida:

#### TYPE

```
marca = ( X, O, ESPACIO);
lugar = (renglón, columna, diagonal);
tablero = ARRAY [1..3, 1..3] OF marca;
```

Se escribirá un procedimiento llamado *ganador* que determine quién ganó y qué tipo de triunfo obtuvo. Tiene un parámetro de arreglo llamado *juego* que es un arreglo bidimensional con un valor de tipo *marca* en cada componente. El procedimiento asignará el valor *X* al parámetro *resultado* si hay tres *X* en fila, *O* si hay tres *O* en fila o *ESPACIO* si no hay ganador. Si se indica un triunfo, el parámetro *tipogane* recibirá el valor *renglón*, *columna* o *diagonal* para indicar el tipo de triunfo.

```
PROCEDURE ganador (VAR juego : tablero;
                    VAR resultado : marca;
                    VAR tipogane : lugar;
(* Determinar si hay ganador, quién es y qué tipo de triunfo *)
(* obtuvo en un juego de gato ya finalizado. *)
V\N
      i : 1..4:                                         (* índice *)
BEGIN
  resultado := ESPACIO;(* suponer que no hay ganador *)
  i := 1;      (* revisar renglones para detectar ganador *)
  WHILE (i <= 3) AND (resultado = ESPACIO) DO
    BEGIN
      IF ((juego[i,1] = X) OR (juego[i,1] = O))
```

```

        AND (juego[i,1] = juego[i,2])
        AND (juego[i,1] = juego[i,3])
THEN BEGIN
    resultado := juego[i,1];
    tipogane := renglon
END
i := i + 1
END;
i := 1; (* revisar columnas *)
WHILE (i <= 3) AND (resultado = ESPACIO) DO
BEGIN
    IF ((juego[1,i] = X) OR (juego[1,i] = O))
        AND (juego[1,i] = juego[2,i])
        AND (juego[1,i] = juego[3,i])
THEN BEGIN
    resultado := juego[1,i];
    tipogane := columna
END;
i := i + 1
END;
IF (resultado = ESPACIO)      (* revisar diagonales *)
    AND ((juego[2,2] = X) OR (juego[2,2] = O))
    AND ((juego[1,1] = juego[2,2]) AND juego[3,3] = juego[2,2]))
        OR ((juego[3,1] = juego[2,2]) AND (juego[1,3] = juego[2,2]))
THEN BEGIN
    resultado := juego[2,2];
    tipogane := diagonal
END
END; (* del procedimiento ganador *)

```

## SECCIÓN 9.5 ANÁLISIS DE EFICIENCIA DE ALGORITMOS [OPCIONAL]

Es probable que el lector ya se haya dado cuenta de que es frecuente que exista más de un algoritmo correcto para resolver un problema dado. Sin embargo, algunos algoritmos correctos son mejores, o más *eficientes*, que otros. ¿Qué significa que un algoritmo sea eficiente? En este capítulo se dijo que, en general, el algoritmo de búsqueda binaria es más eficiente que el algoritmo de búsqueda lineal. En este caso la eficiencia se refiere al *tiempo* requerido para localizar un elemento en un arreglo. No se midió la eficiencia del algoritmo para un caso específico, ya que existen muchos casos en los que una búsqueda lineal es más rápida que una búsqueda binaria. Por ejemplo, si el arreglo es pequeño, las búsquedas lineales son muchas veces muy eficientes. Así, cuando se compara la eficiencia de dos algoritmos, es razonable analizar la eficiencia de cada uno en una situación general. Es decir, se compara el funcionamiento de un algoritmo con datos arbitrarios y se determina cuál es el que funciona mejor en promedio. Ésta no es la única forma de medir la eficiencia; por ejemplo, se podría medir la cantidad de *memoria* que requiere el algoritmo.

En esta sección se analizará la eficiencia de los algoritmos con más detenimiento que la de los programas en computadora. La estimación de la eficiencia de programas ya codificados puede ser una tarea difícil porque depende mucho del sistema de cómputo y lenguaje de programación utilizados. Por otro lado, los algoritmos son independientes de la máquina y se pueden analizar para obtener estimaciones de su eficiencia. De hecho, las ganancias importantes en cuanto a la eficiencia se obtienen sobre algoritmos y no tanto sobre los programas de computadora en sí.

En general, los recursos de tiempo y espacio que requiere la ejecución de un algoritmo dependen del número de datos de entrada del algoritmo. Por ejemplo, en los algoritmos de búsqueda lineal o binaria, el tiempo que se requiere para localizar un elemento y la cantidad de memoria utilizada aumentan conforme crece el tamaño del arreglo de datos de entrada. Así, el rendimiento de un algoritmo dependerá del número de datos de entrada. Supóngase que  $n$  representa el número de datos de entrada. Para analizar la eficiencia de un algoritmo, es preciso resolver el siguiente problema: dados el algoritmo y  $n$  datos de entrada, estímese el tiempo y cantidad de memoria que requiere la ejecución del algoritmo.

¿Cómo se miden los requerimientos de tiempo y memoria de un algoritmo? Los requerimientos de memoria se pueden estimar según la cantidad de memoria que se usa para almacenar elementos tales como variables, constantes y arreglos. En el caso de los algoritmos de búsqueda lineal y binaria, medir la memoria no resulta muy útil, ya que ambos algoritmos utilizan aproximadamente la misma cantidad de memoria. Para medir el tiempo que tarda en ejecutarse un algoritmo para  $n$  datos de entrada se podría contar el número de instrucciones que se van a ejecutar. Sin embargo, muchas de ellas, como son las asignaciones de valores iniciales, requieren muy poco tiempo en comparación con la actividad principal del algoritmo. Por ejemplo, en el caso del algoritmo de búsqueda lineal, la mayor parte del tiempo de ejecución se dedica a comparar el elemento que se debe localizar con los componentes del arreglo. Puesto que lo que interesa es el rendimiento general del algoritmo, es razonable, en el caso del algoritmo de búsqueda lineal, medir su eficiencia mediante el conteo del número de comparaciones.

## Notación O

La medición exacta de la eficiencia de un algoritmo dado es a menudo prácticamente imposible. Por lo general el analista se conforma con una estimación de la eficiencia del algoritmo. Técnicamente, esta estimación se conoce como *orden* del algoritmo. Las estimaciones de eficiencia de los algoritmos se escriben muchas veces mediante la *notación O*. En seguida se verán algunos ejemplos del uso de la notación O.

Supóngase que un algoritmo requiere  $3n^2 + n - 1$  pasos en promedio para ejecutarse con  $n$  datos de entrada. Al crecer mucho el número de datos, el número de pasos ( $3n^2 + n - 1$ ) requeridos para la ejecución del algoritmo se verá dominado por el primer término,  $3n^2$ . Es decir, cuando  $n$  es muy grande,  $3n^2$  es mucho más grande que el otro término,  $n - 1$ . Esto permite despreciar el término  $n - 1$ . El resultado se escribe así:

$$3n^2 + n - 1 = O(n^2)$$

La ecuación dice, en efecto, que la eficiencia del algoritmo no crece más rápidamente que una constante multiplicada por  $n^2$ . En este ejemplo, el orden del algoritmo es  $O(n^2)$ . Obsérvese que si se duplica el número de datos de entrada  $n$ , el número de pasos requeridos para ejecutar el algoritmo aumentará cerca de cuatro veces.

Puede demostrarse que el número promedio de comparaciones que se requiere en una búsqueda lineal con un arreglo de  $n$  elementos es  $n/2$ . Por tanto, el orden de una búsqueda lineal es  $O(n)$ , ya que  $n/2$  no crece con mayor rapidez que una constante multiplicada por  $n$ . De manera similar, se puede demostrar que el algoritmo de búsqueda binaria es de orden  $O(\log_2 n)$  para un arreglo clasificado de  $n$  elementos. Conforme  $n$  crece,  $\log_2 n$  se vuelve significativamente menor que  $n$ . Ésta es la razón de que la búsqueda binaria sea en general mucho más rápida que la búsqueda lineal. Estos resultados se derivan en cursos de computación avanzada.

En la siguiente tabla se muestran algunos órdenes de algoritmos comunes y sus nombres. Con objeto de que el lector aprecie las diferencias importantes entre estos órdenes, la tabla incluye una columna que muestra los tiempos de ejecución de algoritmos representativos de cada uno de los órdenes para  $n = 50$  datos de entrada. Los tiempos suponen el uso de una computadora capaz de ejecutar un paso cada microsegundo ( $10^{-6}$  segundos); es decir, la computadora puede ejecutar un millón de pasos por segundo.

<i>orden</i>	<i>nombre</i>	<i>tiempo de ejecución</i>
$O(1)$	Constante	$10^{-6}$ segundos
$O(\log_2 n)$	Logarítmica	$6 \times 10^{-6}$ segundos
$O(n)$	Lineal	$5 \times 10^{-5}$ segundos
$O(n \log_2 n)$	$n \log n$	$3 \times 10^{-4}$ segundos
$O(n^2)$	Cuadrática	$3 \times 10^{-3}$ segundos
$O(n^3)$	Cúbica	0.13 segundos
$O(2^n)$	Exponencial	36 años
$O(n!)$	Factorial	$10^{51}$ años

Es importante observar, por ejemplo, que si un algoritmo tiene el orden  $O(n^2)$  y otro algoritmo tiene el orden  $O(n^3)$ , el algoritmo con  $O(n^2)$  es más eficiente que el de  $O(n^3)$  si el número de datos de entrada es lo bastante alto. No obstante, es posible que el algoritmo de  $O(n^3)$  sea más eficiente que el de  $O(n^2)$  cuando los datos de entrada son pocos.

Esta sección concluirá con un ejemplo de análisis de eficiencia de un algoritmo para localizar el elemento más grande de un arreglo llamado *lista* con  $n$  elementos. El valor más grande se asignará a la variable *máx*. El algoritmo es, a grandes rasgos, el siguiente:

```

PASO 1 Asignar el valor inicial lista[1] a máx.
PASO 2 Para cada valor entero de i en la escala de 2 a n
    Si máx < lista[i]
        entonces asignar lista[i] a máx

```

Puesto que casi todo el trabajo de este algoritmo se lleva a cabo dentro del ciclo del paso 2, se medirá la eficiencia de este algoritmo mediante el cálculo del número de comparaciones que se hacen dentro del ciclo para localizar el elemento más grande. Cada vez que se ejecuta el cuerpo del ciclo se hace una comparación. El ciclo se ejecuta con valores de  $i = 2, 3, 4, \dots, n$ . Es decir, el cuerpo del ciclo se ejecuta  $n - 1$  veces. Esto implica que el número de comparaciones es  $n - 1$ . El orden del algoritmo es  $O(n)$ , ya que  $n - 1$  no crece con más rapidez que una constante multiplicada por  $n$ . Por tanto, el algoritmo es lineal, lo cual significa, en este caso, que si se duplica el número de datos de entrada, lo mismo sucederá con el número de comparaciones que se requieren para localizar el elemento más grande.

## SECCIÓN 9.6 TÉCNICAS DE PRUEBA Y DEPURACIÓN

Uno de los errores más comunes que se presentan cuando se usan arreglos es que el valor de índice quede fuera de la escala de valores permitidos para el subíndice. Si el índice o subíndice está fuera de la escala, o bien el cálculo no produce el resultado correcto o, en algunos sistemas, se indica un error. He aquí un ejemplo. Supóngase que se desea examinar el arreglo lista para localizar un número llamado *elem* (supóngase que *lista* se declara como ARRAY [1..50] of *integer*). Estudiése el siguiente segmento de programa en Pascal (*índice* es una variable de tipo 1..50):

```
índice := 1;  
WHILE lista[índice] < > elem DO          (* buscar el valor *)  
    índice := índice + 1;
```

Si *elem* está en el arreglo *lista*, la posición de *elem* en el arreglo estará en *índice*. Empero, si *elem* no está en el arreglo *lista*, una vez que el ciclo WHILE pruebe a *lista[50]* el índice subirá a 51 (fuera de la subescala especificada en la declaración), y el ciclo WHILE hará referencia a un elemento que no está en el arreglo (*lista[51]*). Por tanto, es necesario verificar que *índice* esté en la escala apropiada antes de hacer la prueba (“*lista[índice] < > elem*”).

He aquí la solución modificada para el problema anterior:

```
índice := 1;  
WHILE (índice <= 50) AND (lista[índice] < > elem) DO  
    índice := índice + 1;
```

Desafortunadamente, esta “reparación” no resuelve tampoco el problema. La razón es que la expresión booleana de la proposición WHILE se evalúa completamente, sin importar qué valor tenga “(*índice* <= 50)”. Por esto, cuando *índice* adquiere el valor 51, al evaluarse la expresión booleana se hará referencia a *lista[51]*, lo que produce un error. Una forma de evitar el error es verificar por separado el último elemento del arreglo, como se muestra en el siguiente código;

```
índice := 1;  
WHILE (índice < 50) AND (lista[índice] < > elem) DO
```

```

índice := índice + 1;
IF (índice = 50) AND (lista[índice] < > elem)
THEN writeln ('No se encontró el elemento.')
ELSE writeln ('Se encontró el elemento, índice = ', índice)

```

He aquí otra solución que emplea una variable booleana llamada *está* para registrar si se ha localizado el elemento o no.

```

índice := 0;
REPEAT
    índice := índice + 1;
    está := lista[índice] = elem
UNTIL está OR (índice = 50);
IF está
THEN writeln ('Se encontró el elemento, índice = ', índice)
ELSE writeln ('No se encontró el elemento.')

```

Otra causa de errores de índice es la declaración de un número insuficiente de elementos en el arreglo. Por ejemplo, supóngase que se tienen las siguientes declaraciones:

```

TYPE
    lista = ARRAY [1..100] OF integer;
VAR
    calif : lista;

```

El arreglo *calif* almacena cien enteros. Sin embargo, al leer los datos de entrada para asignarle a los elementos del arreglo, es posible que exista un número desconocido de valores seguidos de un fin de archivo. El siguiente segmento en Pascal es representativo del código que se emplea para leer los datos (supóngase que se declara *índice* como variable de tipo *0..100*):

```

índice := 0;
WHILE NOT eof DO
BEGIN
    índice := índice + 1;
    read (calif[índice])
END

```

Si se proporcionan más de cien valores como datos de entrada, el ciclo hará que *índice* sea mayor que cien e intentará colocar un valor en *calif[101]*.

Al utilizar arreglos en Pascal, se recomienda al lector asegurarse de declarar un número de elementos por lo menos igual al número máximo de valores esperados, y además, si se va a leer un número desconocido de valores, emplear una variable que registre el número de valores capturados para verificar que no se tratará de asignar un valor a un elemento de arreglo que no existe.

Declarar arreglos con demasiados elementos normalmente no causa problemas serios, pero los elementos no aprovechados reducen la cantidad de memoria dis-

ponible para otros usos y pueden reducir la velocidad de ejecución del programa. Además, es posible que no se pueda ejecutar el programa en algunos sistemas de memoria limitada. Es prácticamente seguro que la declaración:

#### VAR

```
lista : ARRAY [1..máxint] OF integer;
```

requerirá demasiada memoria para ejecutarse en forma razonable en cualquier sistema de cómputo y normalmente no se ejecutará en absoluto en sistemas de microcomputadora.

He aquí una lista de recordatorios importantes de Pascal que conviene consultar al probar y depurar los programas.

## RECORDATORIOS DE PASCAL

- Las definiciones de arreglos no pueden aparecer en los encabezados de procedimientos o funciones; en vez de ello se utiliza una definición TYPE global.
- Para hacer referencia a los componentes individuales de un arreglo se usa el nombre del arreglo seguido del índice (o lista de índices) encerrado entre paréntesis cuadrados:

```
lista[2]      tabla[2,3]
```

- El índice puede ser cualquier expresión válida que produzca un valor del tipo correcto:

```
lista[2*índice + 1]      tabla[trunc(sqrt(17.3))]
```

- El tipo del índice puede ser cualquier tipo ordinal y el tipo de los componentes puede ser cualquier tipo de datos, incluso un tipo de datos estructurado:

#### TYPE

```
tabla = ARRAY [1..5] OF ARRAY [1..1] OF char;
```

- En las búsquedas lineales se examina el arreglo en orden desde el primer componente hasta que se localiza el elemento deseado o hasta que se examina el último elemento del arreglo.
- En las búsquedas binarias, para localizar un elemento el arreglo clasificado se divide a la mitad en cada paso.
- Los arreglos multidimensionales son arreglos y se pueden declarar en forma abreviada:

#### TYPE

```
nombres = ARRAY [1..50, 1..15] OF char;
```

- Los errores de subíndice constituyen el problema más frecuente cuando se procesan arreglos.
- Las variables que tienen tipos de arreglo idénticos se pueden asignar mutuamente.

**SECCIÓN 9.7 REPASO DEL CAPÍTULO**

En este capítulo se analizaron la definición y la declaración de arreglos tanto de una como de varias dimensiones. Se presentaron muchos segmentos en Pascal para procesamiento de arreglos. También se usaron ciclos FOR y WHILE en abundancia para procesar arreglos. Los arreglos multidimensionales se pueden considerar como arreglo de arreglos.

Muchos problemas que usan arreglos requieren la búsqueda de un elemento en un arreglo o la clasificación de un arreglo en un orden determinado. En este capítulo se presentaron dos técnicas de búsqueda: lineal y binaria. Se presentó en forma detallada un algoritmo de clasificación llamado clasificación por selección. Este algoritmo se aplicó al problema de calcular la mediana de los salarios de los empleados de una compañía. También se resolvió un problema con el uso de arreglos multidimensionales. En una sección opcional del capítulo se comentaron la eficiencia de los algoritmos y la notación O.

En seguida se presenta un resumen de las características de Pascal presentadas en este capítulo que puede servir como referencia en el futuro.

**REFERENCIAS DE PASCAL**

- 1 Arreglo: grupo de elementos del mismo tipo al que se asigna un nombre.

**TYPE nombre = ARRAY [tipo índice] OF tipo componente**

Ejemplo:

**TYPE lista ARRAY [-10..10] OF char;**

- 2 Arreglo multidimensional: Arreglo de arreglos, Ejemplos:

**TYPE**

**tabla = ARRAY [1..5] OF ARRAY [1..10] OF char;**

o bien

**tabla = ARRAY [1..5, 1..10] OF char;**

o bien

**columnas = ARRAY [1..10] OF char;**

**tabla = ARRAY [1..5] OF columnas;**

- 3 Acceso a los componentes de un arreglo: empleese el nombre del arreglo seguido de una lista de expresiones índice encerrada entre paréntesis cuadrados:

**nombrearreglo [ expresión índice ]**

o bien

nombrearreglo [ índice renglón, índice columna ]

Ejemplos:

tabla [3, 5] o tabla [3][5]

lista [2 \* j]

## Avance del capítulo 10

En el siguiente capítulo se estudiará la manipulación de datos no numéricos, incluso la manipulación de secuencias de caracteres. Las computadoras dedica más tiempo a la manipulación de datos no numéricos que a datos de cualquier otro tipo.

## Palabras clave del capítulo 9

arreglo	columna
arreglo clasificado	índice
arreglo multidimensional	notación O
arreglos paralelos	orden
búsqueda binaria	renglón
búsqueda lineal	subíndice
búsqueda secuencial	tipos de datos estructurados
clasificación por selección	tipos de datos simples

## EJERCICIOS DEL CAPÍTULO 9

### ★ EJERCICIOS ESENCIALES

- 1 El producto escalar de dos arreglos unidimensionales de números reales  $a$  y  $b$  definidos así:

VAR

a, b: ARRAY [1..tamarr] OF real;

es la suma de

los valores posibles de  $i$  y  $j$ . Escríbase un procedimiento que calcule la transpuesta de un arreglo de números reales con cinco renglones y cinco columnas.

### ★ ★ EJERCICIOS IMPORTANTES

- 3 Una matriz simétrica  $a$  es un arreglo cuadrado bidimensional con la propiedad de que  $a[i,j] = a[j,i]$ . Una matriz así se puede almacenar en forma compacta (en el arreglo  $ac$ ) ya que basta con almacenar una sola vez los elementos duplicados del arreglo. En primer término, diseñese un método para almacenar un arreglo de éstos sin repetir los elementos duplicados. Después escríbase una función  $OBTSIM(ac, i, j)$  que produzca el valor de  $a[i,j]$  y un procedimiento  $ALMASIM(ac, i, j, val)$  que almacena  $val$  en la localidad apropiada del arreglo compacto  $ac$ . (Sugerencia: basta con almacenar  $n$  elementos de la columna  $n$ ; piénsese en un arreglo unidimensional que contiene los elementos de la columna uno, después de la columna dos y así sucesivamente.)
- 4 Escríbase una función llamada *igual* que tiene como parámetros dos arreglos unidimensionales con el mismo tipo de componentes y subíndices enteros positivos. El primero tiene  $N$  elementos y el segundo tiene  $M$  elementos, donde  $M$  y  $N$  se definen como constantes. Además, *igual* debe determinar si los elementos del segundo arreglo aparecen, de manera contigua y en orden, en algún lugar del primer arreglo. Si así fuera, *igual* debe dar como resultado el índice del primer componente a la izquierda en el primer arreglo donde se presentó la concordancia. En caso contrario, el resultado será cero. Por ejemplo, si el primer arreglo contiene

9 7 12 7 85 2 5

y el segundo arreglo contiene

12 7 85 2

el resultado de *igual* deberá ser tres.

### ★ ★ ★ EJERCICIOS ESTIMULANTES

- 5 Este ejercicio es básicamente igual al ejercicio 4, pero ahora se debe considerar al primer arreglo como si fuera circular, de manera que la concordancia puede continuar desde el final del arreglo hasta el principio. Por ejemplo, si el primer arreglo contiene

3 7 9 1 8

y el segundo arreglo contiene

1 8 3

el resultado de *igual* deberá ser cuatro.

## ★ PROBLEMAS ESENCIALES

- 1 Defínase la distancia entre dos enteros positivos como la suma del valor absoluto de la diferencia entre los dígitos de posiciones correspondientes, MOD 10. Por ejemplo, la distancia entre 417 y 392 es

$$( |4 - 3| + |1 - 9| + |7 - 2| ) \text{ MOD } 10$$

es decir, cuatro. Escríbase un programa que calcule la distancia entre las parejas de enteros que aparezcan como datos de entrada. Tómese nota de que es necesario suponer ceros a la izquierda si los números de una pareja no tienen el mismo número de dígitos.

Ejemplo de entrada:

417 392 5 12

Ejemplo de salida:

La distancia entre 417 y 392 es 4.

La distancia entre 5 y 12 es 4.

- 2 Los datos de entrada son un entero  $N$  y un conjunto de  $N$  caracteres encerrados entre apóstrofos. Averíguese cuántas operaciones de comparación son necesarias entre los elementos de los datos que se van a clasificar para colocar los caracteres en orden ascendente mediante la clasificación por selección.

Ejemplo de entrada:

'D' 'G' 'B'

Ejemplo de salida:

Para colocar los caracteres

D G B

en orden ascendente mediante la clasificación por selección  
se requieren 3 comparaciones.

- 3 Se ha dicho que es preciso examinar en promedio  $N/2$  elementos de un arreglo de  $N$  elementos para determinar si un valor determinado está en el arreglo. El objetivo de este ejercicio es verificar que esto se cumple. Escríbase un procedimiento llamado *buscalineal* que examine un arreglo en busca de un valor que sí está en el arreglo y calcule el número de elementos del arreglo que se examinaron durante la operación de búsqueda. En seguida escríbase un programa que lea  $N$ , un entero, de la primera línea de datos de entrada y  $N$  caracteres únicos de la segunda línea. Cada uno de estos caracteres se debe colocar en un

elemento separado de un arreglo. El programa deberá utilizar entonces *busca-lineal* para determinar el número de comparaciones requeridas para localizar cada uno de los elementos del arreglo de caracteres. En seguida se deberá exhibir el promedio de estos números, así como el valor de  $N/2$ , ambos como números reales. Si la afirmación original es correcta, los dos valores deberán ser aproximadamente iguales. Explíquese la razón de que no sean idénticos los resultados.

Ejemplo de entrada:

6  
ABFECD

Ejemplo de salida:

El número promedio de comparaciones requeridas fue 3.5  
El valor de  $N / 2$  es 3.0

### ★ ★ PROBLEMAS IMPORTANTES

- 4 Dados los mismos datos del problema 2, realícese una clasificación por selección del arreglo, pero manipúlese solamente un segundo arreglo que contenga los índices de los elementos. Si se supone que el arreglo  $d$  contiene los datos que se van a clasificar, entonces un segundo arreglo  $x$  tendrá los índices de los componentes del arreglo  $d$ . Después de la clasificación por selección,  $x[1]$  tendrá el índice en  $d$  del componente más pequeño y  $x[n]$  tendrá el índice en  $d$  del componente más grande. Es decir, después de la clasificación,  $d[x[1]]$  será el valor del componente más pequeño y  $d[x[n]]$  será el valor del componente más grande. Exhíbase el arreglo  $x$ .

Ejemplo de entrada:

4 'G"A'I"C'

Ejemplo de salida:

El arreglo índice contiene  
2 4 1 3

- 5 Los datos de entrada no son más de cien parejas de enteros. Cada pareja debe tratarse como un componente y clasificarse de tal manera que el primer entero de cada pareja esté en orden ascendente. Si los primeros componentes de varias parejas tienen el mismo valor, se deberá hacer una clasificación secundaria de manera que los valores de los segundos componentes estén en orden descendente.

Ejemplo de entrada:

12 41  
9 304

12 63  
8 12  
9 512

Ejemplo de salida:

8 12  
9 512  
9 304  
12 63  
12 41

- 6 La **convolución** de un arreglo  $D$  de componentes reales con otro arreglo  $F$  de componentes reales se define como la suma de los productos escalares (véase el ejercicio 1) de  $F$  con cada grupo de componentes adyacentes de  $D$ . Por ejemplo, si  $D$  contiene

1 7 3 4 2

y  $F$  contiene

0.2 0.6 0.2

entonces

$$\begin{aligned}\text{Convolución} &= 1 * 0.2 + 7 * 0.6 + 3 * 0.2 \\&\quad + 7 * 0.2 + 3 * 0.6 + 4 * 0.2 \\&\quad + 3 * 0.2 + 4 * 0.6 + 2 * 0.2 \\&= 5.0 + 4.0 + 3.4 \\&= 12.4\end{aligned}$$

Escríbase un programa en Pascal que tenga como datos de entrada dos enteros  $M$  y  $N$  seguidos de  $M$  números reales que forman el arreglo  $D$ , y  $N$  números reales que forman el arreglo  $F$ . Calcúlese la convolución y exhibase el resultado.

Ejemplo de entrada:

5 3 1.0 7.0 3.0 4.0 2.0 0.2 0.6 0.2

Ejemplo de salida:

La convolución es 1.2400000e + 01

### ★ ★ ★ PROBLEMAS ESTIMULANTES

- 7 Este problema está relacionado con el problema 6. Se desea calcular también la convolución, pero esta vez se define como la suma de todos los productos escalares que se pueden formar entre  $D$  y  $F$ , incluso la “continuación” del fi-

nal hacia el principio de  $D$ . Si se usan los mismos datos que en el problema 6, se incluirá también

$$4 * 0.2 + 2 * 0.6 + 1 * 0.2 \\ + 2 * 0.2 + 1 * 0.6 + 7 * 0.2$$

con lo que la convolución tendrá el valor 17.0. Los formatos de entrada y de salida son los mismos que para el problema 6.

- 8 Los datos de entrada incluyen dos enteros positivos,  $M$  y  $N$ , no mayores de diez, seguidos por los  $M$  renglones de un arreglo de enteros con  $M$  renglones y  $N$  columnas. Esto va seguido de un entero  $P$  y  $P$  enteros adicionales. Escribáse un programa que localice una concordancia entre los  $P$  enteros que se leen al último y cualquier secuencia de enteros contiguos en el arreglo de  $M$  por  $N$  en cualquier renglón, columna o diagonal, hacia adelante o hacia atrás. Si no se puede encontrar la concordancia, exhibáse un mensaje apropiado. En caso contrario, exhibáanse los índices del componente del arreglo  $M$  por  $N$  en el que comienza la concordancia y la dirección en la que se encontró la concordancia. (Un programa todavía más difícil es localizar todas las concordancias posibles.)

### Ejemplo de entrada:

3	4		
4	7	9	3
6	6	2	0
3	1	2	3
3			
1	2	3	

### Ejemplo de salida:

La concordancia comienza en el renglón 3 columna 2 en diagonal hacia arriba, en dirección a la derecha.

o bien

La concordancia comienza en el renglón 3, columna 2 en renglón, en dirección a la derecha.

(En el caso del problema más difícil se exhibirían ambos mensajes.)

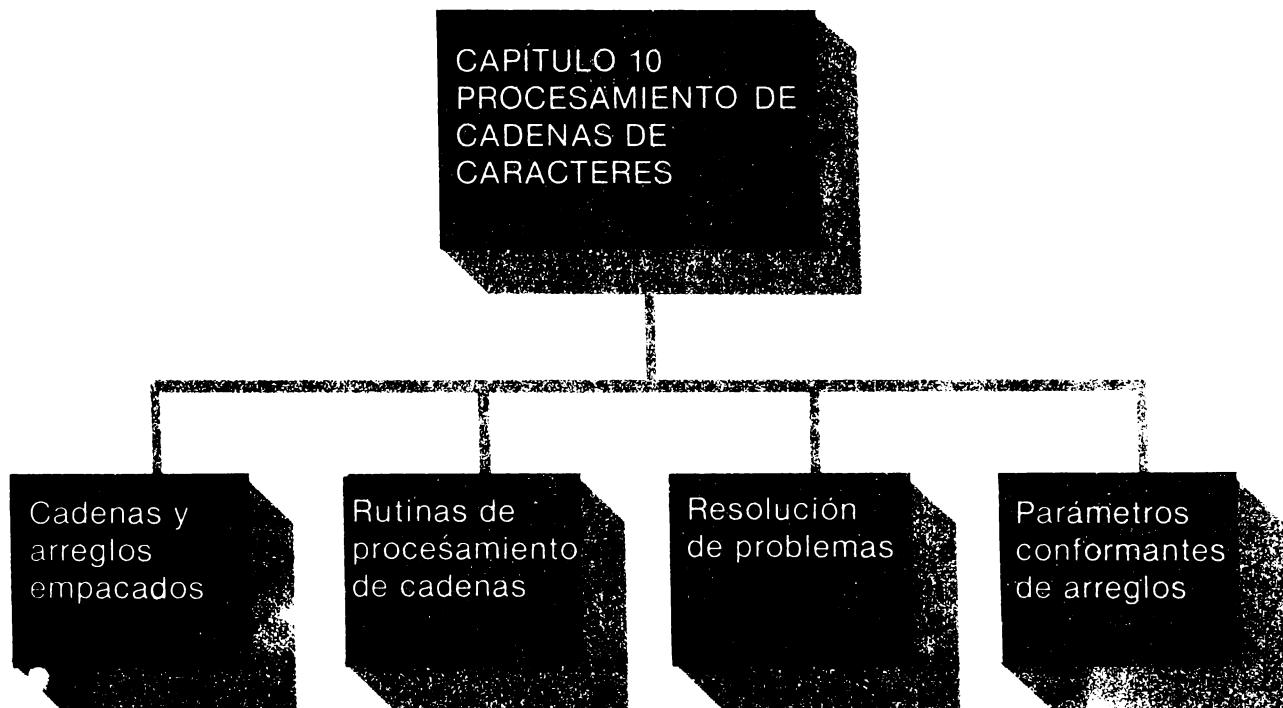
- 9 Los datos de entrada incluyen parejas de líneas. Cada línea contiene la representación de un entero que (probablemente) es mayor que *máxint*. Exhibase cada una de las parejas de enteros y su suma. El número de dígitos en un solo entero no será mayor de 50.

### Ejemplo de entrada:





# CAPÍTULO 10



## PROCESAMIENTO DE CADENAS DE CARACTERES

## OBJETIVOS

Después de completar este capítulo, el lector deberá ser capaz de:

- Reconocer y aplicar los arreglos empacados
- Reconocer y aplicar las variables de cadena
- Reconocer y aplicar los procedimientos estándar *pack* y *unpack*
- Escribir rutinas en Pascal estándar para procesar cadenas
- Resolver, probar y depurar problemas que usen cadenas
- Reconocer y aplicar los parámetros conformantes de arreglo a las operaciones con cadenas [opcional]

## PANORAMA GENERAL DEL CAPÍTULO

En los albores de la programación de computadoras (década de 1950) las computadoras se usaban principalmente para procesar datos numéricos. Sin embargo, en los últimos veinte años, el procesamiento de datos no numéricos se ha convertido en una fuente importante de aplicaciones de las computadoras. Los editores de textos, traductores de lenguajes, programas amables con el usuario y grandes bases de datos formadas por datos de caracteres son sólo unos cuantos de los ejemplos de procesamiento de datos no numéricos. En este capítulo se hará hincapié en el procesamiento de cadenas de caracteres. En particular, se demostrará que la forma más efectiva de emplear cadenas de caracteres y variables de cadenas es declararlas como *arreglos empacados* de caracteres. Las variables de cadena se pueden exhibir como un solo objeto mediante una sola proposición *write*, y no carácter por carácter como sucede en el caso de los arreglos de caracteres no empacados. Otra ventaja de las variables de cadena es la posibilidad de compararlas mediante los operadores relacionales. Además, se analizarán dos procedimientos estándar en Pascal que permiten alternar entre los formatos de datos empacados y no empacados: *pack* y *unpack*.

Desafortunadamente, Pascal estándar no incluye funciones o procedimientos para manipular cadenas. En la sección 10.2 se presentan técnicas para escribir rutinas de procedimiento de cadenas. Se aplicarán algunas de las técnicas a un problema de edición de textos en la sección 10.3.

El capítulo termina con una sección opcional sobre parámetros conformantes de arreglo que tienen aplicación en las operaciones de cadenas<sup>1</sup>. Los parámetros conformantes de arreglo permiten procesar parámetros de arreglo de longitud variable, cosa que puede ser muy efectiva y útil al escribir procedimientos de proceso de cadenas.

<sup>1</sup>Los parámetros conformantes de arreglo se incluyen en el estándar ISO del lenguaje Pascal, pero no en el estándar ANSI. Casi todos los sistemas de Pascal estándar incluyen arreglos conformantes.

En el último capítulo se mostró la forma de declarar un arreglo de caracteres. Por ejemplo, la declaración

```
TYPE
  palabra = ARRAY [1..6] OF char;
VAR
  nombre = palabra;
```

declara un arreglo de seis caracteres. Si se usa esta declaración, se requerirán seis proposiciones de asignación y seis localidades de memoria para almacenar el nombre *ALICIA*, con una localidad por carácter (véase la Fig. 10-1).

### Arreglos empacados

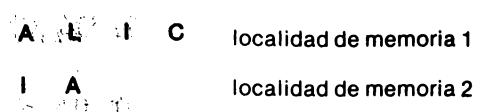
En el caso de muchos sistemas de cómputo, lo anterior implica un desperdicio de memoria, sobre todo si el arreglo de caracteres es muy grande. Por tanto, el Pascal permite declarar *arreglos empacados* de caracteres. Esto indica al sistema de Pascal que debe guardar tantos caracteres como sea posible en cada localidad de memoria (o *palabra*). El número de caracteres empacados en una localidad de memoria varía según el sistema de cómputo usado. En las minicomputadoras se pueden empacar normalmente dos caracteres por palabra, pero en sistemas más grandes es posible almacenar cuatro caracteres en cada palabra. De manera que si se usa la declaración

```
TYPE
  palabra = PACKED ARRAY [1..6] OF char;
VAR
  nombre = palabra;
```

en vez de la declaración anterior, la proposición de asignación requerirá únicamente dos palabras, en vez de seis, en un sistema de cómputo que empaque cuatro caracteres en una palabra. La figura 10-2 muestra la palabra *ALICIA* empacada en dos palabras. Obsérvese que basta incluir la palabra reservada PACKED (em-

**Figura 10-2** Almacenamiento de un arreglo de caracteres.

- A      localidad de memoria 1
- L      localidad de memoria 2
- I      localidad de memoria 3
- C      localidad de memoria 4
- I      localidad de memoria 5
- A      localidad de memoria 6



**Figura 10-2** Arreglo empacado.

pacado) antes de la palabra reservada **ARRAY** (arreglo). Los arreglos empacados de caracteres cuyo subíndice inferior es uno se conocen también como **cadenas de caracteres**. Las variables que se declaran como arreglos empacados de caracteres se llaman **variables de cadena**. Ya se han analizado las cadenas de caracteres conocidas como **constantes de cadena**. Por ejemplo, la proposición

**writeln ('Comienza la ejecución')**

incluye la constante de cadena ‘Comienza la ejecución’ que se almacena como arreglo empacado.

Una ventaja del uso de arreglos empacados es que las constantes o variables de cadena se pueden exhibir directamente, como se hizo con la constante de cadena anterior, en vez de carácter por carácter. La siguiente proposición **write** exhibe el nombre contenido en la variable *nombre*, ya declarada como arreglo empacado:

**write (nombre)**

Sin la definición de arreglo empacado sería preciso escribir el contenido de *nombre* carácter por carácter, como lo hace el siguiente segmento de código:

```
FOR i = 1 TO 6 DO
    write (nombre[i])
```

Los elementos de los arreglos empacados se pueden leer de la misma forma que los arreglos no empacados, un componente a la vez. Por ejemplo, el Pascal estándar acepta lo siguiente para el arreglo empacado *nombre*:

```
FOR i := 1 TO 6 DO
    read (nombre[i])
```

Sin embargo, algunas versiones no estándar no permiten que un componente de un arreglo empacado aparezca como parámetro de **read** o **readln**. En vez de ello, es necesario leer los datos y asignarlos a una variable no empacada, para después asignar el valor obtenido al componente del arreglo empacado. Por ejemplo, supóngase que se desea leer quince caracteres de una línea de datos de entrada (se supondrá que hay por lo menos quince) y después exhibirlos. El siguiente es el código usual que se debe emplear para capturar arreglos empacados:

```
TYPE
  palabra = PACKED ARRAY [1..15] OF char;
```

```

VAR
    índice : integer;           (* índice del arreglo      *)
    primeras15 : palabra;      (* variable de arreglo empacado   *)
    uncar : char;              (* un carácter no empacado      *)
FOR índice : 1 TO 15 DO
BEGIN
    read (uncar);
    primeras15[índice] := uncar
END
writeln (primeras15)

```

Tómese en cuenta que la proposición *writeln* exhibe el contenido del arreglo empacado directamente y sin usar subíndices. Más adelante, en esta sección, se mostrará la forma de usar el procedimiento estándar *pack* para copiar un arreglo no empacado como un arreglo empacado sin procesar cada carácter individualmente en un ciclo. Al asignar un valor a un arreglo empacado completo mediante una proposición de asignación, el valor debe tener exactamente el mismo número de caracteres que la variable. Por ejemplo, si se trata de una constante, el número de caracteres entre los apóstrofos debe ser exactamente el mismo que tiene el arreglo empacado que representa a la variable de cadena:

```

VAR
    cadena1 : PACKED ARRAY [1..6] OF char;
    cadena2 : PACKED ARRAY [1..8] OF char;
    cadena3 : PACKED ARRAY [1..6] OF char;
cadena1 := 'Pascal';
cadena2 := 'Estándar';

```

Es común agregar espacios en blanco adicionales al final de una constante de cadena si se quiere asignar a una variable de cadena con más caracteres:

```
cadena2 := 'Poco ';
```

Las variables de cadena se pueden asignar a otras variables de cadena, pero se debe respetar la misma restricción. Así, la proposición de asignación

```
cadena3 := cadena1
```

es correcta, pero la proposición de asignación

```
cadena2 := cadena1
```

no lo es. Nótese que las variables de cadena no se rellenan automáticamente con espacios en blanco cuando se les asigna el valor de un arreglo empacado o constante de cadena con un número menor de elementos. También es importante recordar que las variables de cadena *deben* tener un subíndice inferior de uno.

Otra ventaja importante del uso de las cadenas de caracteres en forma de arreglos empacados es que se pueden comparar directamente mediante los opera-

dores relacionales ( $=$ ,  $<$ ,  $>$ ,  $=$ ,  $<=$ , y  $<>$ ). La alternativa es comparar los caracteres de las cadenas uno por uno, lo que sería una tarea realmente tediosa.

Las relaciones que se cumplen entre las cadenas de caracteres están representadas en los siguiente ejemplos:

'GATO'	<	'LEÓN'
'vaca'	>	'gato'
'KENNEDY'	>	'JOHNSON'
'PARTICIPAN'	<	'PARTICIPEN'

Los espacios en blanco y las letras mayúsculas y minúsculas pueden dificultar la comprensión de las comparaciones de cadenas. En la mayor parte de los sistemas, el carácter del espacio en blanco precede a todos los caracteres alfabéticos en las comparaciones. Por tanto, la siguiente relación se cumple en esos sistemas:

'JOHNSON' > 'JOHN'

No obstante, se recomienda tener cuidado al mezclar letras mayúsculas y minúsculas en las comparaciones. Si el sistema emplea el ASCII (véase el apéndice F), la siguiente relación se cumplirá:

'León' < 'gato'

Por otro lado, si el sistema emplea el EBCDIC, se cumple lo contrario:

'gato' < 'León'

El ordenamiento de las cadenas es similar al ordenamiento de las palabras en un diccionario. Al comparar cadenas, la computadora compara los valores ordinales (los que produciría la función *ord*) de los caracteres en las posiciones correspondientes en cada cadena de izquierda a derecha, uno por uno, hasta que sea posible determinar la relación entre las cadenas. En tanto el carácter de una cadena sea exactamente igual al carácter correspondiente de la otra cadena, la comparación continuará. Si todos los caracteres correspondientes son idénticos, las cadenas son iguales. En caso contrario, la relación entre la primera pareja de caracteres que no sean idénticos define la relación que existe entre las dos cadenas.

Piénsese, por ejemplo, en la comparación entre 'KENNEDY' y 'JOHNSON'. La comparación se realiza como se muestra en seguida. También puede observarse la relación que existe entre cada pareja de caracteres.

K	E	N	N	E	D	Y
>	<	>	=	<	<	>
J	O	H	N	S	O	N
↑						

Ésta es la única comparación que se hace.

Tan pronto como la computadora determina que los primeros caracteres de cada una de las cadenas no son idénticos ('K' > 'J'), la relación entre las cadenas

queda determinada. Nótese que ninguna de las relaciones entre los caracteres a la izquierda de la primera desigualdad pueden afectar la relación entre las cadenas. Por tanto, 'AZZZZ' es menor que 'ZAAAA', ya que 'A' es menor que 'Z'. Las cadenas que tienen diferentes longitudes no se pueden comparar directamente. Por ejemplo, no se debe escribir

**IF 'ABCD' < 'ABC' (\* incorrecto \*)**

Una forma de comparar cadenas de diferentes longitudes es agregar espacios en blanco al final de la cadena más corta para que tenga la misma longitud que la cadena más larga. La comparación anterior se podría escribir así:

**IF 'ABCD' < 'ABC' (\* correcto, rellenando con espacios \*)**

Hasta aquí se ha visto que las variables de cadena (declaradas como arreglos empacados de caracteres con subíndice inferior igual a uno) tienen las siguientes ventajas con respecto a los arreglos no empacados:

- 1 Las cadenas se pueden exhibir como un solo objeto mediante una proposición *write* o *writeln*, en vez de la exhibición carácter por carácter que es necesaria cuando se usan arreglos no empacados.
- 2 Las cadenas se pueden comparar directamente, lo que suele requerir un tiempo de ejecución menor que la comparación de arreglos de caracteres no empacados.
- 3 En algunos sistemas, las cadenas ahorran espacio en la memoria.

Desafortunadamente, el uso de variables de cadena tiene ciertas desventajas:

- 1 Aunque los arreglos empacados completos se pueden usar como parámetros, no puede hacerse lo mismo con los componentes de los arreglos empacados. Sin embargo, es posible asignar un componente de un arreglo empacado a un componente de un arreglo no empacado, o a una variable simple, y viceversa. Por tanto, el componente empacado se puede copiar en una variable simple, transferirse como parámetro y finalmente copiarse de nuevo en el componente empacado después de la invocación del procedimiento.
- 2 Si el programa requiere conversiones frecuentes entre arreglos empacados y no empacados, el tiempo de ejecución puede ser mayor que si no se usaran arreglos empacados.

Para decidir si se deben usar arreglos no empacados o sus contrapartes empacadas deben tenerse en cuenta muchos factores: el volumen de datos que se ha de manipular, la memoria disponible, los requisitos de tiempo de ejecución y el sistema de Pascal específico que se utilice, todo esto puede influir en la elección. Algunas versiones de Pascal pueden almacenar datos empacados y no empacados en el mismo formato, especialmente cuando el número de caracteres por palabra es uno. En estos casos, el uso de arreglos empacados en lugar de sus equivalentes no empacados no parece tener alguna ventaja obvia. No obstante, si existen ventajas. Si se desea que los programas sean transportables, debe ser fácil pasarlos de

un sistema de cómputo a otro. Si el sistema en Pascal que se usa originalmente no requería datos empacados, el programador se ve tentado a omitir la palabra PACKED y hacer caso omiso de las restricciones que imponen los arreglos empacados. En un caso así, si el programa se pasa a una computadora diferente que sí tiene las restricciones de los arreglos empacados, se invertirá mucho tiempo en la modificación del programa para que cumpla con los requisitos. Esto se debe a las características no estándar del Pascal: siempre que sea posible, debe evitarse su uso.

También es posible empacar otros arreglos en Pascal. Por ejemplo,

**TYPE**

```
estudio = PACKED ARRAY [1..128] OF Boolean;
```

podría ahorrar bastante espacio. No obstante, las cadenas son los datos estructurados empacados que se encuentran con más frecuencia.

### Procedimientos estándar de Pascal: *pack* y *unpack*

Otra forma de transferir datos a un arreglo empacado es copiar el contenido de un arreglo no empacado directamente mediante el procedimiento estándar *pack*. Supóngase, por ejemplo, que se desea leer los primeros quince caracteres de una línea de datos de entrada (como en un ejemplo anterior). Si se emplean las declaraciones

**TYPE**

```
palabra = PACKED ARRAY [1..15] OF char;
nopalabra = ARRAY [1..15] OF char;
```

**VAR**

```
empacanom : palabra;
nombre : nopalabra;
índice : integer;
```

el segmento en Pascal

```
FOR índice := 1 TO 15 DO
    read (nombre[índice])
pack (nombre, 1, empacanom)
```

colocará los primeros quince caracteres en la variable de arreglo empacado *empacanom*. El procedimiento de empacado realiza de hecho las mismas acciones que

```
FOR índice := 1 TO 15 DO
    empacanom[índice] := nombre[índice]
```

pero requiere una codificación menos explícita.

El procedimiento estándar *pack* (empacar) tiene la forma general

```
pack (arreglo1, inicio, arreglo2)
```

donde *arreglon* es el nombre de un arreglo no empacado, *inicio* es una expresión (del mismo tipo que el índice de *arreglon*) y *arregloe* es un arreglo empacado (cuyos componentes son del mismo tipo que los de *arreglon*). En esencia, el procedimiento de empacar hace que los componentes del arreglo no empacado *arreglon* se asignen, en orden y a partir del elemento especificado mediante *inicio*, a los componentes del arreglo empacado *arregloe*. Se llenan todos los componentes del arreglo empacado, por lo que es necesario que existan suficientes componentes en el arreglo no empacado después del componente indizado mediante *inicio* (inclusive) para llenar el arreglo empacado. Por ejemplo,

**pack (nombre, 2, empacanom)**

no se podrá ejecutar, ya que solamente existen catorce componentes entre *nombre[2]* y *nombre[15]* y se requieren quince para llenar el arreglo empacado *empacanom*.

El procedimiento estándar *unpack* sirve para realizar la función opuesta. En la invocación

**unpack (arregloe, arreglon, inicio)**

los parámetros *arregloe*, *arreglon* e *inicio* tienen la misma interpretación que en el procedimiento de empacado, pero el efecto es que todos los componentes del arreglo empacado *arregloe* se copien, uno por uno, en el arreglo no empacado *arreglon*, comenzando por el índice *inicio*. Una vez más, es preciso que existan por lo menos tantos componentes desde la posición *inicio* hasta el fin del arreglo no empacado como hay en el arreglo empacado; en caso contrario se presentará un error. Para invertir la operación de empacado

**pack (arreglon, 1, arregloe);**

se escribirá

**unpack (arregloe, arreglon, 1);**

## EJERCICIOS DE LA SECCIÓN 10.1

1 Determíñese cuáles asignaciones son válidas dada la declaración

**TYPE**

```
palabra = PACKED ARRAY [1..15] OF char;
enunciado = PACKED ARRAY [1..80] OF char;
```

**VAR**

```
palabra1, palabra2 : palabra;
frase : enunciado;
```

- a) palabra1 := palabra2
- b) palabra1 := frase

- c) palabra1[1] := 'A';
- d) frase[1] := palabra1[15]
- e) palabra2 := 'La palabra vale'

- 2 De las siguientes ventajas, ¿cuál *no* corresponde a los arreglos empacados?
- a) El tiempo que requiere la computadora para tener acceso a un elemento determinado es más corto que en el caso de un arreglo normal.
  - b) Los arreglos empacados del mismo tipo se pueden comparar directamente.
  - c) Es posible que el arreglo ocupe menos memoria en la computadora.
  - d) A los arreglos empacados de caracteres se les puede asignar una cadena de la misma longitud.
  - e) Los arreglos empacados de caracteres se pueden emplear como parámetros de los procedimientos *write* y *writeln*.
- 3 Escríbese una función llamada *comparar* que produzca el valor —1 si un arreglo empacado de veinte caracteres, llamado *palabra1*, precede a un arreglo similar llamado *palabra2*; que produzca el valor cero si *palabra1* = *palabra2* y que produzca el valor 1 si *palabra1* sigue a *palabra2*. Inclúyanse las definiciones de tipo apropiadas.

- 4 Examíñese la siguiente función:

```

TYPE
  palabra = PACKED ARRAY [1..80] OF char;
FUNCTION algo(VAR unapal: palabra): integer;
CONST
  blanco = ' ';
VAR
  temp : integer;
BEGIN
  temp := 81;
  REPEAT
    temp := temp - 1
  UNTIL (unapal[temp] <> ' ') OR (temp = 1);
  IF (temp = 1) AND (unapal[1] = blanco)
  THEN algo := 0
  ELSE algo := temp
END;

```

¿Puede el lector describir lo que calcula esta función?

- 5 ¿Qué exhibe el siguiente procedimiento si el parámetro verdadero es *unapal* = 'lacsap'?

```

TYPE palabra = PACKED ARRAY [1..6] OF char;
PROCEDURE qué (VAR unapal : palabra)
VAR
  tempc : char;
  temp : ARRAY [1..6] OF char;
  índice : integer;
BEGIN
  unpack (unapal, temp, 1)
  FOR índice := 1 TO 3 DO
    BEGIN
      tempc := temp[índice];
      temp[índice] := temp[7—índice];
      temp[7—índice] := tempc
    END;
    pack(temp, 1, unapal);
    writeln ('La palabra es', unapal)
  END;

```

## SECCIÓN 10.2 PROCESAMIENTO DE CADENAS

En esta sección se examinarán varios problemas comunes que se presentan al manipular datos de cadenas de caracteres. Pascal estándar no incluye funciones o procedimientos para manipulación de cadenas, y aunque algunas versiones pueden contener funciones no estándar con ese propósito, hasta la fecha no existen funciones estándar para llevar a cabo esas tareas.<sup>2</sup>

El primer problema se referirá a la captura del nombre completo de una persona a partir de los datos de entrada y su colocación en un arreglo empacado.

### Problema 10.1

*Escribáse un procedimiento en Pascal llamado leenombre que capture el nombre completo de una persona y lo almacene en un arreglo empacado. El número máximo de caracteres del nombre completo no pasará de 50 y el final del nombre irá seguido inmediatamente de un fin de línea en los datos de entrada. Si el nombre no tiene una longitud exacta de 50 caracteres, los caracteres que queden al final del arreglo empacado deben ser espacios en blanco. Por ejemplo, la línea de datos de entrada*

**Francisco I. Madero <eoln >**

*produciría la asignación al arreglo empacado del valor*

**Francisco I. Madero**

<sup>2</sup>Es posible que en el futuro se modifiquen los estándares del lenguaje de programación Pascal para incluir un nuevo tipo de datos, operadores y procedimientos o funciones estándar que permitan la manipulación de cadenas de caracteres.

que contiene exactamente 50 caracteres. Si el nombre contiene más de 50 caracteres, se deberá truncar el excedente y omitirlo.

El siguiente procedimiento resuelve correctamente este problema; se supone que ya se ha hecho la siguiente definición global:

**TYPE**

```
cadena = PACKED ARRAY [1..50] OF char;
```

**PROCEDURE** leenombre (VAR nomcomp : cadena);

(\* Leer un nombre completo terminado por <eoln> y asignarlo a nomcomp \*)  
VAR

```
j, índice: integer;  
uncar : char;
```

**BEGIN**

```
índice := 0;
```

```
WHILE NOT eoln AND (índice < 50) DO
```

**BEGIN**

```
índice := índice + 1;
```

```
read (uncar);
```

```
nomcomp[índice] := uncar
```

**END;**

(\* Llenar de espacios los caracteres no usados, si los hay \*)

```
FOR j := índice + 1 to 50 DO
```

```
nomcomp[j] := ' ';
```

**readln**

**END;**

Supóngase ahora que se desea leer una lista de nombres, uno por línea, y asignarlos a un arreglo de cadenas. Se supondrá que la lista de nombres termina con un fin de archivo y que no se van a procesar más de cien nombres. El siguiente procedimiento realiza esta tarea, siempre que se hayan definido globalmente los tipos

**TYPE**

```
cadena = PACKED ARRAY [1..50] OF char;
```

```
lista = ARRAY [1..100] OF cadena;
```

La magnitud del parámetro se ajustará según el número de nombres que realmente se lean y almacenen en el arreglo.

**PROCEDURE** leelista (VAR tabla : lista; VAR tamarr : integer);

(\* Leer una lista de no más de cien nombres que termina con un fin \*)

(\* de archivo, con un nombre por línea y no más de 50 caracteres \*)

(\* por nombre. El número de nombres leídos se colocarán en la \*)

(\* variable tamarr. \*)

**BEGIN**

```
tamarr := 0;
```

```

WHILE NOT eof AND (tamarr < 100) DO
BEGIN
    tamarr := tamarr + 1;
    leenombre (tabla[tamarr])
END;

```

**Problema 10.2**

*Escríbese un programa en Pascal que lea una lista de nombres y más tarde los exhiba en el orden opuesto al que se leyeron.*

Se supone que los procedimientos *leenombre* y *leelista* son los que ya se escribieron. El programa se podrá escribir entonces así:

```

PROGRAM invertir (input, output);
(* Leer una lista de no más de cien nombres de 50 caracteres o menos *)
(* Exhibirlos después en el orden opuesto al que se leyeron      *)
TYPE
    cadena = PACKED ARRAY [1..50] OF char;
    lista = ARRAY [1..100] OF cadena;
VAR
    tabla: lista;          (* nombres en el orden en que se leen *)
    númnom : integer;      (* número de nombres *)
    nomínd : integer;      (* índice del nombre que se exhibe *)
PROCEDURE leenombre ...   (* como se definió antes *)
PROCEDURE leelista        (* como se definió antes *)
(* Programa principal *)
BEGIN
    leelista (tabla, númnom);           (* leer los nombres *)
    FOR nomínd := númnom DOWNTO 1 DO
        writeln (tabla[nomínd]);        (* exhibir un nombre *)
END.

```

El funcionamiento del programa es muy sencillo, una vez que se dispone de los procedimientos *leenombre* y *leelista*. En seguida se presenta un ejemplo de los datos de entrada y la salida que se obtiene al ejecutarse el programa.

Jorge Washington <eoln >	Mao Tse Tung
Alberto Einstein <eoln >	Winston Churchill
Winston Churchill <eoln >	Alberto Einstein
Mao Tse Tung <eoln >	Jorge Washington
<eof>	

El siguiente problema extraerá y exhibirá el nombre de pila de cada uno de los nombres completos de una lista.

**Problema 10.3**

*Escríbase un procedimiento en Pascal para exhibir el primer nombre de cada uno de los nombres completos que aparezcan en la lista que se pasa como parámetro.*

El siguiente procedimiento realiza esta tarea. El procedimiento supone que el primer espacio en blanco que se encuentre será el que separa al primer nombre del apellido. Supóngase que ya se hicieron las declaraciones de los problemas anteriores y que el nombre de pila tiene un máximo de quince caracteres.

```

PROCEDURE extraer (VAR tabla : lista; tamarr : integer);
(* Extraer y exhibir el primer nombre de cada nombre completo      *)
(* de tabla[1] a tabla[tamarr], pasando el tamaño de la             *)
(* lista (tamarr) como parámetro de valor.                         *)
CONST
    espacios = '                 '; (* 15 espacios en blanco *) 
TYPE
    palabra = PACKED ARRAY [1..15] OF char;
VAR
    nompila : palabra;          (* un nombre de pila      *)
    uncar : char;               (* un carácter de un nombre *)
    j,      (* subíndice del nombre completo que se procesa      *)
    índice: integer;            (* subíndice del nombre de pila      *)
BEGIN
    FOR j := 1 TO tamarr DO      (* con cada nombre      *)
        BEGIN
            nompila := espacios;   (* dar valor inicial      *)
            índice := 0;           (* no hay caracteres todavía *)
            uncar := tabla[j, 1];   (* obtener primera letra  *)
            WHILE (uncar <> ' ') AND (índice < 15) DO
                BEGIN
                    índice := índice + 1      (* obtener otro carácter *)
                    nompila[índice] := uncar; (* guardarlo *)
                    uncar := tabla[j, índice + 1] (* obtener siguiente carácter *)
                END;
                writeln (nompila);      (* exhibir el nombre de pila *)
        END;
END;

```

Si se invoca el procedimiento con los mismos datos de entrada que se usaron en el ejemplo anterior, la salida será la esperada:

Jorge  
Alberto  
Winston  
Mao

La **concatenación** es una muy común operación que junta dos cadenas, extremo con extremo, para formar una sola. Por ejemplo, supóngase que se tienen las siguientes cadenas:

`cadena1 = 'Hoy es'`  
`cadena2 = 'el día antes de mañana.'`

La concatenación de las dos cadenas sería

`'Hoy es el día antes de mañana.'`

He aquí otros ejemplos de concatenación.

<i>cadena1</i>	<i>cadena2</i>	<i>concatenación</i>
'Tom '	' Jones'	'Tom Jones'
'Yo soy '	'aquél'	'Yo soy aquél'
'Calle'	'25'	'Calle25'
'ABC'	'DEF'	'ABCDEF'
'A'	'?'	'A?'

El siguiente problema consiste en escribir un procedimiento que concatene dos cadenas.

#### Problema 10.4

*Escríbase un procedimiento para concatenar dos cadenas. Los parámetros formales deberán ser las dos cadenas y sus longitudes, junto con la cadena concatenada y su longitud. Supóngase que todas las cadenas tienen una longitud máxima de 80 caracteres y verifíquese que la cadena producida no tenga más de 80 caracteres.*

El siguiente procedimiento resuelve el problema, siempre que se haya hecho esta definición global de tipo

**TYPE**

`cadena = PACKED ARRAY [1..80] OF char;`

`PROCEDURE juntar (VAR cadena1, cadena2, resultado : cadena;`  
`largo1, largo2 : integer;`  
`VAR largor : integer);`

(\* Concatenar cadena1 y cadena2 (cuyas longitudes respectivas son \*)  
(\* largo1 y largo2 caracteres) para formar resultado. Largo1 más \*)  
(\* largo2 deberá ser menor o igual que 80. \*)  
`VAR`

```

índice: integer;
BEGIN
  IF largo1 + largo2 > 80
  THEN writeln ('ERROR en juntar: resultado > 80 caracteres.')
  ELSE BEGIN
    resultado := cadena1; (* copiar cadena1 *)
    (* agregar cadena2 a resultado *)
    FOR índice := 1 TO largo2 DO
      (* copiar un carácter de cadena2 en resultado *)
      resultado[largo1 + índice] := cadena2[índice];
    largo := largo1 + largo2
  END
END;

```

Las rutinas de manipulación de cadenas que se han presentado hasta ahora requieren que la longitud máxima de las cadenas sea un valor constante. Esto se debe al requisito de que los parámetros formales de arreglo deben tener exactamente el mismo tipo que los parámetros verdaderos. Más adelante, en este capítulo se presentarán rutinas generales de manipulación de cadenas que utilizan únicamente Pascal estándar y permiten manipular cadenas cuyas longitudes máximas difieren.

## EJERCICIOS DE LA SECCIÓN 10.2

- 1 Concaténense las siguientes cadenas:
  - a) '2 + 2' y '= 4'
  - b) 'La hora' y 'ha llegado'
  - c) 'Ave. Bolívar' y '2500'
  - d) 'Nueva York' y ', Nueva York'
  - e) 'blanco' y 'blanco'
- 2 Escríbase una función llamada *ctablancos* que dé como resultado el número de espacios en blanco en un arreglo empacado de 80 caracteres llamado *enunciado*.
- 3 ¿Cuáles de los siguientes métodos leerán y almacenarán caracteres en un arreglo empacado y llenarán el resto de la cadena con espacios en blanco? El fin de la cadena estará indicado mediante el fin de línea en los datos de entrada. Supóngase que el tipo del arreglo es el siguiente:

```

CONST
  máx = 20;
  blanco = ' ';
TYPE
  tipocad = PACKED ARRAY [1..máx] OF char;
VAR
  cad : tipocad;
  i : integer;
  total : real;

```

- a) read (cad);  
 b) i := 1;  
 WHILE (i < máx) AND (NOT eoln) DO  
     read (cad[i]);  
 WHILE (i = máx) DO  
     CAD[i] := blanco;  
 c) FOR i := 1 TO máx DO  
     read (cad[i]);  
 d) i := 1;  
 WHILE (i <= máx) AND (NOT eoln) DO  
   BEGIN  
     read (cad[i]);  
     i := i + 1  
 END;  
 WHILE (i = máx) DO  
   BEGIN  
     cad[i] := blanco;  
     i := i + 1  
 END;  
 e) FOR i := :1 TO máx DO  
   IF NOT eoln  
     THEN read (cad[i]);

- 4 ¿Qué exhibirá el siguiente procedimiento si el parámetro de entrada *unapal* es ‘JUAN PÉREZ’?

```

TYPE palabra = PACKED ARRAY [1..10] OF char;
PROCEDURE quién (VAR unapal : palabra);
VAR
  cuenta : integer;
BEGIN
  FOR cuen'ta := 1 TO 5 DO
    unapal [cuenta] := ' ';
  writeln (unapal);
  FOR cuenta := 1 TO 5 DO
    unapal [cuenta] := unapal [cuenta + 5]
  writeln (unapal);
  FOR cuenta := 1 TO 5 DO
    unapal [cuenta + 5] := ' ';
  writeln (unapal)
END;
```

- 5 Escríbase un procedimiento similar al procedimiento *extraer* que extraiga el apellido de cada uno de los nombres completos en una lista. Supóngase que los nombres constan exclusivamente de nombre de pila y apellido:

Indira Ghandi

En esta sección se resolverá un problema que depende considerablemente de las cadenas de caracteres. La definición del problema es:

### Problema 10.5

*La Compañía de Sistemas Transportables se especializa en la creación de programas que se pueden emplear en diversos sistemas de cómputo. La empresa quisiera desarrollar un editor de textos sencillos que sea transportable. Por tanto, ha decidido escribir el programa en Pascal estándar. El editor de textos deberá ser capaz de:*

- Leer un archivo de texto línea por línea, suponiendo un máximo de 72 caracteres por línea y un máximo de 500 líneas
- Borrar cualquier número de líneas del texto
- Insertar cualquier número de líneas en el texto
- Exhibir o listar cualquier número de líneas junto con sus números de línea
- Sustituir una línea por otra nueva
- Explicar al usuario cuáles son los comandos de que dispone
- Terminar su ejecución.

*Escribase el programa en Pascal estándar que realice todas estas tareas.*

Es posible subdividir este problema en varios subproblemas:

SUBPROBLEMA 1: Leer el texto.

SUBPROBLEMA 2: Leer un comando.

SUBPROBLEMA 3: Procesar el comando.

El subproblema 1 se puede resolver mediante técnicas similares a las que se presentaron en la sección anterior para capturar una lista de nombres completos de los datos de entrada.

El subproblema 2 leerá comandos legales. Se abreviarán los comandos empleando exclusivamente su inicial: B para borrar, I para insertar, L para listar, S para sustituir, A para pedir ayuda y T para terminar.

El subproblema 3 procesará el comando mediante una proposición CASE que seleccione las acciones. Cada uno de los comandos hará que se ejecute un procedimiento apropiado. Los comandos de borrar, insertar, listar y sustituir deberán hacer referencia a números de línea. Irán seguidos de dos enteros,  $m$  y  $n$ , para indicar las líneas que se van a manipular. Esta tabla describe las acciones del editor en respuesta a ejemplos de comandos.

comando	acción del editor
B 2 3	Borrar líneas de la 2 a la 3.
I 3 5	Insertar 3 líneas antes de la línea 5.
L 2 5	Listar líneas de la 2 a la 5
S 5 10	Las siguientes seis líneas de entrada sustituyen a las líneas 5 a 10.

El algoritmo del editor de textos se expresa mediante el siguiente seudocódigo.

*Algoritmo de editor de textos*

```
Capturar el texto.  
Capturar un comando.  
Mientras el comando no es T (terminar):  
    Si no es el comando H, leer los enteros m y n  
    En caso que el comando sea:  
        B: Borrar líneas m y n.  
        I: Insertar m líneas antes de línea n.  
        L: Listar líneas m a n.  
        S: Sustituir líneas m a n.  
        H: Exhibir breve resumen de comandos.  
        Otro: Exhibir mensaje de error apropiado.  
    Capturar siguiente comando.
```

El programa principal incluirá las siguientes definiciones de tipo:

```
CONST  
    tamarchmáx = 500;  
TYPE  
    cadena PACKED ARRAY [1..72] OF char;  
    bloque = ARRAY [1..tamarchmáx] OF cadena;
```

Para capturar el texto, se leerá línea por línea, y se buscará ya sea un carácter de fin de línea o un máximo de 72 caracteres. Esto se repite hasta que se llegue al fin del texto o se lea el número máximo de líneas permitido en el archivo. Se supondrá que el fin del texto está indicado mediante una línea que comienza con dos puntos ('..'). Esta línea no es parte del texto, es un centinela.

El procedimiento *leetexto* realiza esta tarea, con lo que se resuelve el subproblema 1. Este procedimiento llama a otro, *leelinea*, para leer cada una de las líneas. Esta organización es parecida a la que se usó para procesar los nombres completos de un grupo de individuos.

```
PROCEDURE leelinea (VAR línueva : cadena);  
(* Leer línea que termina con <eoln> y asignarla a línueva *)  
VAR  
    cuentacar : integer;  
    uncar : char;  
BEGIN  
    FOR cuentacar := 1 TO 72 DO      (* inicial a la línea *)  
        línueva[cuentacar] := ' '; (* asignar espacios como valor *)  
    cuentacar := 0;  
    WHILE NOT eoln AND (cuentacar < 72) DO  
    BEGIN  
        cuentacar := cuentacar + 1;  
        read (uncar);
```

```

línueva[cuentacar] := uncar
END;
readIn
END

PROCEDURE leetexto (VAR archtexto : bloque; VAR numlíneas : integer);
(* Leer archivo de texto existente (hasta "..") y contar líneas. *)
VAR
    listo : Boolean;
BEGIN
    numlíneas := 0;
    listo := false;
    WHILE NOT listo AND (numlíneas < tamarchmáx) DO
        BEGIN
            numlíneas := numlíneas + 1;
            leelínea (archtexto[numlíneas]);
            (* Revisar si se leyó centinela de fin de texto, "..". *)
            IF (archtexto[numlíneas,1] = '.' AND
                (archtexto[numlíneas,2] = '.'))
            THEN BEGIN
                listo := true;
                numlíneas := numlíneas - 1 (* no contar .. *)
            END
        END
    END;

```

En primer término se codificará el procedimiento de borrado. La mejor forma de explicarlo es con un ejemplo. Supóngase que se tienen diez líneas de texto. Se desea borrar de la línea dos a la cinco. Una forma de hacerlo es desplazar las líneas de la seis a la diez hacia arriba (hacia la línea uno) cuatro líneas, de manera que la línea seis se convierta en la línea dos, la línea siete se convierta en la línea tres, y así sucesivamente. La última línea sería ahora la línea seis, ya que se borraron cuatro líneas de las diez originales.

En general, el borrado de las líneas  $m$  hasta  $n$  hace que las líneas que siguen al texto borrado se desplacen hacia arriba sobre el texto borrado. Las líneas que se desplazan van de  $n + 1$  hasta  $tamarch$ . Por último, se le resta a  $tamarch$  ( $n - m + 1$ ), que es el número de líneas borradas.

Los parámetros del procedimiento son el arreglo que contiene las líneas de texto, los números de línea que se deben borrar ( $m$  y  $n$ , donde  $m \leq n$ ), y el número de líneas que tiene actualmente el archivo ( $tamarch$ ).

```

PROCEDURE borrar (VAR archtexto: bloque; VAR tamarch: integer;
                   m, n : integer);
(*Borrar líneas m a n del texto, donde m<= n<= tamarch. *)
VAR
    índice, j : integer;
BEGIN
    (* Desplazar hacia arriba las líneas de n + 1 hasta la última *)

```

```

(* n — m + 1 líneas *)
FOR índice := n + 1 TO tamarch DO
BEGIN
    j := índice — (n + 1);
    archtexto[m + j] := archtexto[índice]
END;
(* Calcular nuevo tamaño del archivo. *)
tamarch := tamarch — (n — m + 1)
END;

```

El procedimiento de inserción también se puede explicar con un ejemplo. Se usa el comando I 2 5 para pedir la inserción de dos líneas antes de la línea cinco. Esto requiere que las líneas de la cinco hasta la última (*tamarch*) se desplacen hacia abajo dos líneas, de manera que la línea cinco se convierta en la siete, la línea seis se convierta en la ocho y así sucesivamente. Las líneas nuevas se convertirán en las líneas cinco y seis.

En general, para insertar *m* líneas antes de la línea *n*, se desplazarán hacia abajo las líneas de *n* hasta *tamarch* *m* líneas, se insertarán las *m* líneas nuevas de texto y por último se calculará otra vez *tamarch* mediante la suma de *m*. Cabe hacer notar que el desplazamiento debe comenzar con la última línea y no con la primera. De lo contrario, la segunda línea que se ha de desplazar quedará destruida antes de que se desplace.

```

PROCEDURE insertar (VAR archtexto: bloque; VAR tamarch: integer;
                    m, n: integer);
(* Insertar m líneas antes de la línea n del texto, donde      *)
(* m + tamarch < = tamarchmáx y n < = tamarch.      *)
VAR
    índice : integer;
BEGIN
    (* Desplazar hacia abajo líneas de n hasta la última, m líneas *)
    FOR índice := tamarch DOWNTO n DO
        archtexto[índice + m] := archtexto[índice];
    (* Capturar m líneas e insertarlas en el archivo *)
    FOR índice := 1 TO m DO
        leelinea (archtexto[n + índice - 1]);
    (* Calcular nuevo tamaño del archivo. *)
    tamarch := tamarch + m
END;

```

El procedimiento de listado exhibirá las líneas de la *m* a la *n*, suponiendo que *m* <= *n*. Las líneas irán precedidas de su número de línea.

```

PROCEDURE listar (VAR archtexto: bloque; m, n: integer);
(* Listar líneas de la m a la n, donde m < = n < tamarch      *)
(* precedidas de los numeros de linea.      *)
VAR
    índice : integer;

```

BEGIN

```
FOR índice := m TO n DO
    writeln (índice:3, ': ', archtexto[índice])
END;
```

El procedimiento de sustitución cambiará las líneas de la *m* a la *n* por el mismo número de líneas nuevas.

```
PROCEDURE sust (VAR archtexto : bloque; m, n : integer);
(* Sustituir líneas de la m a la n por líneas nuevas,   *)
(* donde m <= n <= tamarch.      *)
VAR
    índice : integer;
BEGIN
    (* Llamar leelinea repetidamente para capturar líneas nuevas. *)
    FOR índice := m TO n DO
        leelinea (archtexto[índice])
END;
```

El procedimiento *ayuda* no tendrá parámetros y simplemente exhibirá un mensaje breve para explicar los comandos disponibles.

```
PROCEDURE ayuda;
(* Exhibir descripción breve de comandos disponibles. *)
BEGIN
```

```
writeln ('Comandos disponibles:');
writeln ('      B  m  n  Borrar de la línea m hasta la n');
writeln ('      I  m  n  Insertar m líneas antes de la n');
writeln ('      L  m  n  Listar de la línea m hasta la n');
writeln ('      S  m  n  Sustituir líneas de la m a la n');
writeln ('      A          Exhibir este mensaje de ayuda ');
writeln ('      T          Terminar ejecución')
END;
```

El programa principal es relativamente sencillo. Basta con capturar comandos, validar los parámetros y llamar a los procedimientos apropiados para llevar a cabo la edición.

```
PROGRAM editor (input, output);
(* *****)
(* Editor de textos sencillo con los siguientes comandos:      *)
(* B m n Borrar de la línea m hasta la n                      *)
(* I m n Insertar m líneas antes de la n                      *)
(* L m n Listar de la línea m hasta la n                      *)
(* S m n Sustituir líneas de la m a la n                      *)
(* A      Exhibir un resumen de los comandos                  *)
```

(\* T Terminar ejecución \*)  
 (\* "m" y "n" son enteros positivos y tienen algunas \*)  
 (\*) restricciones, como se indica en el código. El archivo \*)  
 (\*) original que se va a editar aparece como datos de entrada ter- \*)  
 (\*) minados por una línea que comienza con dos puntos. \*)  
 (\*\*\*\*\*\*)

## CONST

tamarchmáx = 500; (\* número máximo de líneas \*)

## (\* texto de los mensajes de error \*)

mensaje0 = 'Este comando no existe.';  
 mensaje1 = 'Error. Primer núm. de línea mayor que el segundo.';  
 mensaje2 = 'Error. Se excedería tamaño máximo del archivo.';  
 mensaje3 = 'Error. Primer núm. de línea excede tamaño archivo.';  
 mensaje4 = 'Error. Segundo núm. de línea excede tamaño archivo.';  
 mensaje5 = 'Error. Número de línea negativo.'

## TYPE

cadena = PACKED ARRAY [1..72] OF char;  
 bloque = [1..tamarchmáx] OF cadena;

## VAR

archtexto : bloque;  
 tamarch : integer;  
 comando : char;  
 m, n : integer;

## (\* Aquí se deben insertar todos los procedimientos que se mencionan \*)

PROCEDURE leelinea ...  
 PROCEDURE leetexto ...  
 PROCEDURE borrar ...  
 PROCEDURE insertar ...  
 PROCEDURE listar ...  
 PROCEDURE sust ...  
 PROCEDURE ayuda ...  
 (\* Programa principal \*)

## BEGIN

writeln ('Editor');  
 writeln ('Oppri. la A para obtener ayuda');  
 leetexto (archtexto, tamarch);  
 write ('Comando: ');  
 read (comando);  
 WHILE comando < > 'T' DO (\* ¿Comando de terminación? \*)  
 BEGIN  
 (\* No, probar si es válido otro comando. \*)  
 IF comando IN ['B', 'I', 'L', 'S', 'A']  
 THEN CASE comando OF  
 (\* comando de borrado \*)

'D' : BEGIN:

readln (m,n);  
 IF m > n

```

THEN writeln (mensaje1)
ELSE IF n > tamarch
    THE writeln (mensaje4)
ELSE IF m < 1
    THEN writeln (mensaje5)
    ELSE borrar (archtexto, tamarch, m, n)
END;

(* comando de inserción *)
'I' : BEGIN
    readln (m,n);
    IF tamarch + m > tamarchmáx
        THEN writeln (mensaje2)
    ELSE IF n > tamarch
        THEN writeln (mensaje3)
    ELSE IF n < 1
        THEN writeln (mensaje5)
        ELSE insertar (archtexto, tamarch, m,n)
    END;
(* comando de listado *)
'L' : BEGIN
    readln (m,n);
    IF m > n
        THEN writeln (mensaje1)
    ELSE IF m > tamarch
        THEN writeln (mensaje5)
    ELSE IF m < 1
        THEN writeln (mensaje5)
        ELSE listar (archtexto, m, n)
    END;
(* comando de sustitución *)
'S' : BEGIN
    readln (m,n);
    IF m > n
        THEN writeln (mensaje1)
    ELSE IF n > tamarch
        THEN writeln (mensaje3)
    ELSE IF m < 1
        THEN writeln (mensaje5)
        ELSE sust (archtexto, m, n)
    END;
(* comando de ayuda *)
'H' : BEGIN
    ayuda;
    readln (* pasar por alto fin de línea *)
END

```

```

END (* del CASE *)
ELSE (* si no es 'B', 'I', 'L', 'S'. o 'A' *)
BEGIN
  writeln (mensaje0)
  readln (* pasar por alto parámetros, si los hay, y eoln *)
  END
  write ('Comando: ');
  read (comando)
END
END.

```

---

## SECCIÓN 10.4 PARÁMETROS CONFORMANTES DE ARREGLO Y CADENAS [OPCIONAL]

Muchas veces se critica al Pascal por su falta de procedimientos o funciones integradas (estándar) para manipular cadenas de caracteres. Por tanto, muchas versiones de Pascal tienen extensiones no estándar para el procesamiento de cadenas. Esto es lamentable por dos razones: la primera es que hay muy poca o ninguna estandarización de estas funciones. Un programador que use alguna de estas funciones no estándar no podrá pasar su trabajo de una máquina a otra sin invertir una cantidad considerable de esfuerzo. La segunda razón es que Pascal estándar *sí* incluye características que permiten el procesamiento de cadenas.

El “secreto” del procesamiento de cadenas de caracteres en Pascal estándar es el parámetro conformante de arreglo. Esta característica permite escribir procedimientos que pueden tener parámetros de arreglo con escalas de índice variables. Más adelante, en esta sección, se examinará un conjunto de procedimientos que pueden realizar varias operaciones estándar sobre cadenas de caracteres de longitud variable y diferentes tamaños máximos.

### Parámetros conformantes de arreglo

Como ya sabe el lector, es posible pasar arreglos completos como parámetros a los procedimientos y funciones, siempre que los parámetros verdaderos tengan exactamente el mismo tipo que los parámetros formales. Así, el tamaño del arreglo del parámetro verdadero debe concordar con el tamaño del parámetro formal. Por ejemplo, supóngase que un procedimiento llamado *leecadena* lee una cadena de 80 caracteres. Las declaraciones y encabezado de procedimiento apropiados serían los siguientes:

```

TYPE
  cadena = PACKED ARRAY [1..80] OF char;
VAR
  línea: cadena;
PROCEDURE leecadena (VAR unalín : cadena);

```

Para invocar el procedimiento se escribirá normalmente

*leecadena (línea);*

Supóngase que se hicieron también estas declaraciones:

```
TYPE
    chica = PACKED ARRAY [1..25] OF char;
VAR
    partelinea : chica;
```

No es posible utilizar el procedimiento *leecadena* del modo como se formuló originalmente para leer *partelinea* porque *partelinea* no tiene exactamente el mismo tipo que el parámetro formal de *leecadena*, *unalín*. Con las características de Pascal que se han estudiado hasta ahora, sería necesario escribir varios procedimientos para leer cadenas, uno para cada longitud de cadena que se desee capturar. Es obvio que ésta no es una solución muy buena, sobre todo porque el Pascal estándar permite una mejor.

La solución que proporciona Pascal estándar es permitir que los límites de los índices de los parámetros formales de arreglo sean variables. Entonces el parámetro de arreglo efectivo incluirá no sólo el arreglo, sino también los límites del índice. Ahora se verá un ejemplo para el procedimiento *leecadena*. El nuevo encabezado de procedimiento podría ser

```
PROCEDURE leecadena
    (VAR unalín : PACKED ARRAY [inicio..final:integer] OF char);
```

Ahora se podría invocar el procedimiento *leecadena* tanto con *leecadena (línea)* como con *leecadena (partelinea)*. En la primera de estas invocaciones se asignarían los valores 1 y 80, respectivamente, a los límites *inicio* y *final* que necesita *leecadena*, y en la segunda invocación los límites serían 1 y 25.

Son pocas las restricciones que se aplican al uso de parámetros conformantes de arreglo. La primera restricción es que sólo se puede empacar la última dimensión de un arreglo multidimensional. Es obvio que esto no representa un problema real. La segunda restricción es que el número de dimensiones, el tipo de componentes y el tipo base de los índices del arreglo deben ser los mismos. Así, es posible emplear los límites de subíndice que se deseen para el parámetro verdadero siempre que los límites del subíndice del parámetro formal tengan el mismo tipo base. Las demás restricciones son similares a las que se podrían esperar en el caso de otros tipos de parámetros. Por ejemplo, no se puede emplear un arreglo no empacado como parámetro verdadero si el parámetro formal correspondiente está empacado, y viceversa.

La sintaxis exacta de los parámetros conformantes de arreglo está dada en el diagrama de sintaxis del apéndice A. En general, estos parámetros se forman al escribir el nombre del parámetro formal, un signo de dos puntos y la palabra ARRAY o PACKED ARRAY seguida de los límites entre paréntesis cuadrados. Los límites se representan mediante dos identificadores separados por “..”, seguidos de un signo de dos puntos y el tipo base de los límites. Para representar pa-

rámetros de arreglos multidimensionales, los límites se repiten (dentro de los paréntesis cuadrados) separados mediante signos de punto y coma. Por ejemplo, cualquier arreglo bidimensional no empacado de números reales con subíndices enteros sería aceptable como parámetro verdadero del procedimiento *resolver* cuyo encabezado es

#### PROCEDURE resolver

```
(mat : ARRAY [bajo1..alto1 : integer; bajo2..alto2 : integer] OF real);
```

El tipo de los parámetros formales parece ser complejo, pero en realidad es muy sencillo. Los límites de la primera dimensión son *bajo2* y *alto1* (enteros) y los límites de la segunda dimensión son *bajo1* y *alto2* (también enteros). Si se quisiera calcular la suma de los números reales contenidos en el parámetro de arreglo, se escribiría

```
suma := 0;
FOR índice1 := bajo1 TO alto1 DO
    FOR índice2 := bajo2 TO alto2 DO
        suma := suma + mat[índice1, índice2];
```

donde *suma* es una variable real e *índice1* e *índice2* son variables enteras.

### Operaciones con cadenas

Como ilustración del uso de parámetros conformantes de arreglo se escribirá una serie de procedimientos y funciones para manipulación de cadenas que pueden ser muy útiles. De hecho, son pocos los compiladores de Pascal no estándar que incluyen más funciones que las que se presentan aquí. En particular, se examinarán procedimientos y funciones que realicen las siguientes tareas:

- Copiar una constante o variable de cadena en otra cadena (procedimiento *copcad*).
- Crear una cadena de longitud cero (procedimiento *cadnul*).
- Determinar la longitud de una constante o variable de cadena (función *loncad*).
- Concatenar dos cadenas (procedimiento *concat*).
- Comparar dos cadenas (función *compcad*).
- Extraer una subcadena de una cadena dada (procedimiento *subcad*).
- Calcular la posición de la primera ocurrencia de una cadena dada en otra cadena (función *índice*).
- Determinar el primer carácter de una cadena que no es también carácter de una segunda cadena (función *verif*).
- Leer una línea y almacenarla en una cadena (función *leelin*).
- Exhibir el contenido de una cadena (procedimiento *poncad*).

Es conveniente permitir que varíen dos atributos que tienen las cadenas. El primero de ellos es el número máximo de caracteres de la cadena y el segundo es el

número verdadero de caracteres de la cadena. Cuando se declara una variable de cadena como

## VAR

```
unacad : PACKED ARRAY [1..40] OF char;
```

se sabe que la cadena siempre tendrá 40 caracteres y que solamente se puede pasar a procedimientos y funciones mediante parámetros del mismo tipo. Si se usan parámetros conformantes de arreglos será posible usar diferentes valores para el número máximo de caracteres de la cadena y seguir empleando los mismos procedimientos y funciones con todas. Empero, haber declarado el arreglo de manera que pueda almacenar 40 caracteres no significa que siempre se tendrá 40 caracteres válidos. Por ejemplo, supóngase que se usa una cadena para almacenar el apellido de una persona. Si se especifica un límite superior de 40 caracteres para la variable, se estarán limitando los apellidos a 40 caracteres, aunque la mayor parte de los apellidos requieren menos. ¿Cómo se indica que los caracteres finales no son realmente parte del nombre?

Para hacer esto, es preciso incluir ya sea explícita o implícitamente una especificación de longitud en las cadenas de caracteres. Para indicar la longitud de manera implícita se escogería un carácter que normalmente no se use (por ejemplo *chr(0)* en muchas máquinas) y se hace que aparezca inmediatamente después del último carácter “verdadero” de los datos. Si se almacenara el nombre ‘Zarkov’ en el arreglo de 40 caracteres con especificación implícita de longitud, se tendría la siguiente disposición de caracteres:

'Z'	'a'	'r'	'k'	'o'	'v'	<i>chr(0)</i>	?	?	?
1	2	3	4	5	6	7	8	9	...

Los caracteres de las posiciones 8 a 40 en este arreglo se muestran como signos de interrogación porque no se sabe, y no importa, qué caracteres están almacenados allí. El terminador especial, *chr(0)*, indica que los caracteres que siguen a esa posición no son realmente parte de la cadena.

La especificación de longitud explícita requiere reservar uno o más de los caracteres de la cadena para almacenar la longitud. Por ejemplo, si se usa el primer carácter de la cadena para almacenar la longitud, ‘Zarkov’ aparecerá así:

<i>chr(6)</i>	Z	a	r	k	o	v	?	...	?
1	2	3	4	5	6	7	8	...	40

Obsérvese que la longitud se almacena como carácter cuyo valor ordinal es el número de caracteres “verdaderos” de la cadena.

Ambas técnicas requieren por lo menos un carácter adicional en la cadena para indicar la longitud. Además, el enfoque implícito requiere que no se emplee el carácter terminador especial como carácter de datos. La estrategia explícita limita el número de caracteres de la cadena al valor ordinal más grande permitido para un

carácter (normalmente 255) a menos que se consuman dos o más caracteres de la cadena para almacenar su longitud. Sin embargo, el lector debe darse cuenta de que éstas son las técnicas, usadas en versiones no estándar de Pascal y otros lenguajes de programación, que sirven para indicar la longitud de las variables de cadena, de manera que el Pascal estándar permite al programador escoger cuál de las dos técnicas desea emplear.

En los procedimientos y funciones que siguen, se ha elegido usar un carácter terminador implícito. El valor ordinal que se usa queda especificado por la constante global *terminador*. En el resto de esta sección se presentará un análisis de cada uno de los procedimientos y funciones, seguido de un programa completo que los aprovecha para resolver un problema real. El código de los procedimientos y funciones aparece en el programa.

### **Procedimiento *copcad***

Si se desea copiar una variable o constante de cadena en una segunda variable de cadena, se puede usar el procedimiento *copcad*. Si la cadena que se debe copiar es más larga que el número de caracteres permitidos en la cadena resultante, se perderán los caracteres sobrantes.

### **Procedimiento *cadnul***

En Pascal no se permite escribir “”; es decir, no existe una constante de cadena de longitud cero. El lector se preguntará de qué puede servir una cadena de longitud cero. En el problema que se encuentra al final de esta sección se verá un ejemplo de una cadena así, pero en general se puede usar una cadena de caracteres nula, o de longitud cero, de manera similar a como se emplea el cero en la suma aritmética. Por ejemplo, si se concatena la cadena nula y una cadena *C*, el resultado es siempre la misma cadena *C*. La cadena nula indica que ninguno de los caracteres de la variable de cadena es significativo. El procedimiento *cadnul* permite a una variable de cadena representar a la cadena nula mediante el almacenamiento del carácter terminador en la primera posición del parámetro de arreglo empacado.

### **Función *loncad***

Esta función permite determinar el número de caracteres que se usan verdaderamente en una variable de cadena. Para hacerlo, cuenta los caracteres que preceden al terminador y producen esta cuenta como resultado.

### **Procedimiento *concat***

Ya se examinó la operación básica de concatenación de cadenas, y el procedimiento *concat* realiza la misma operación con cadenas de caracteres de longitud variable. El algoritmo empleado es muy parecido al que se presentó anteriormente.

## Función *compcad*

La función *compcad* permite comparar dos variables de cadena de caracteres aunque no tengan las mismas longitudes reales o máximas. El código compara explícitamente los caracteres de las dos cadenas, uno por uno, hasta haber procesado todas las parejas de caracteres o encontrar una pareja desigual de caracteres.

## Procedimiento *subcad*

El procedimiento *subcad* permite extraer un número arbitrario de caracteres desde cualquier posición inicial en una variable de cadena. Esto permite, por ejemplo, examinar fragmentos de una cadena de caracteres y, una vez localizados, almacenar la subcadena que interesa en una variable de cadena de carácter separada con el fin de continuar su procesamiento. También puede usarse para eliminar una parte de la cadena mediante la extracción y almacenamiento en otra variable de la parte de la cadena que se desea conservar.

## Función *índice*

La función *índice* requiere dos variables de cadena de caracteres como parámetros. Si se les llama *busca* y *halla*, el resultado será la posición en *busca* donde aparece por primera vez *halla*. Por ejemplo, supóngase que *busca* contiene ‘MISSISSIPPI’ y *halla* contiene ‘SS’. En este caso ‘índice (*busca*, *halla*)’ producirá el valor entero tres, ya que la cadena ‘SS’ aparece por primera vez a partir de la posición tres en ‘MISSISSIPPI’. Si la cadena *halla* no aparece en *busca*, *índice* produce cero. *Índice* se usa con frecuencia para localizar espacios en blanco en una cadena de palabras con el fin de extraer las palabras individuales mediante *subcad*.

## Función *verif*

La función *verif* se usa a menudo para realizar la operación opuesta a *índice*. También requiere dos argumentos, una cadena *buscar* que se examinará y una cadena *noestá* que contiene los caracteres que se deben pasar por alto en *buscar*. En esencia, *verif* da como resultado un entero que indica la primera posición en *buscar* ocupada por un carácter que no está en *noestá*. Si todos los caracteres de *buscar* aparecen también en algún lugar de *noestá*, *verif* dará como resultado cero. Por ejemplo, supóngase que se desea localizar el primer carácter que no sea espacio en blanco en una cadena. En ese caso *noestá* tendrá un solo carácter, ‘ ’. Si todos los caracteres de la cadena examinada son espacios en blanco, el resultado de *verif* será cero. En caso contrario, el resultado será la posición del primer carácter que no es un espacio en blanco.

## Función *leelín*

La función *leelín* capturará una línea completa de datos de entrada y la almacenará en una variable de cadena que se proporciona como parámetro (efecto secundario).

## Procedimiento *poncad*

El procedimiento *poncad* exhibirá los caracteres de su parámetro único en el archivo de salida. No se escribirá un fin de línea, de manera que es posible colocar varias cadenas en una sola línea de salida.

## Problema completo: tabulación de uso de palabras

Con el fin de ilustrar el uso de estos procedimientos y funciones, se escribirá un programa completo en Pascal que producirá una lista alfabetizada de las palabras que se emplean en un documento, el cual se proporciona como datos de entrada. Además, calculará el número de veces que aparece cada palabra. Para hacerlo más sencillo, se definirá una palabra como cualquier secuencia de caracteres que no incluya espacios en blanco y separada de otras palabras mediante espacios en blanco, tabulaciones o caracteres de fin de línea. En el capítulo 6 se presentó un problema similar a éste, aunque no se registraron o alfabetizaron las palabras.

---

```

PROGRAMA clasifpal (input, output);
(* Lee un texto de longitud variable y produce una lista clasificada    *)
(* de todas las palabras (secuencias de caracteres delimitadas por      *)
(* espacios, tabulacion^s o eoln ) y su frecuencia de aparición.        *)
CONST
  terminador = 0;(* adecuado para computadoras que usan ASCII *)
  máxpal = 100;          (* número máximo de palabras únicas *)
  lonmáxpal = 20;         (* longitud máxima de las palabras + 1 *)
  lonmáxlin = 81;         (* longitud máxima de las líneas + 1 *)
TYPE
  palabra = PACKED ARRAY [1..lonmáxpal] OF char;
VAR
  palabras : ARRAY [1..máxpal] OF palabra;      (* las palabras *)
  cuentas : ARRAY [1..máxpal] OF integer;        (* las cuentas *)
  línea : PACKED ARRAY [1..lonmáxlin] OF char;   (* línea de entrada *)
  unapal : palabra;                            (* última palabra capturada *)
  blanco: PACKED ARRAY [1..2] OF char;          (* un espacio en blanco *)
  númpal : integer;                           (* número de palabras únicas *)
(* Copcad (b,a) copia una cadena (a) en una variable de cadena    *)
(* (b). Si por su longitud a no cabe en b, se le truncará.        *)
(* Obsérvese que este procedimiento funciona con una constante    *)
(* de cadena o con una variable de la forma apropiada.           *)
PROCEDURE copcad
  VAR cadsal: PACKED ARRAY [salbaj..salalt: integer] OF char;
  cadent: PACKED ARRAY [entbaj..entalt] OF char);

```

## VAR

```

i,j : integer;
car : char;
más : Boolean;

BEGIN
  i := entbaj;
  j := salbaj;
  más := true;
  (* Copiar de cadent[i] a cadsal[j] en tanto j sea menor      *)
  (* o igual que la longitud de cadsal e i sea menor o        *)
  (* igual que la longitud de cadent y no se haya copiado      *)
  (* el terminador.                                              *)
WHILE (más AND (i <= entalt) AND (j <= salalt)) DO
BEGIN
  cadsal[j] := cadent[i];
  IF ord(cadsal[j])<>terminador
  THEN BEGIN
    i := i + 1;
    j := j + 1
    END
  ELSE más := false (* ya se copió el terminador      *)
  END;
  IF más (* no se ha copiado el terminador      *)
  THEN IF j <= salalt
    THEN cadsal[j] := chr(terminador) (* no se trunca *)
    ELSE cadsal[salalt] := chr(terminador) (* se trunca *)
END: (* de copcad *)
(* Dado que es imposible representar una cadena vacía como      *)
(* constante en PASCAL, se emplea un procedimiento especial      *)
(* para almacenar cero caracteres en una cadena                  *)
PROCEDURE cadnul
  (VAR cadsal : PACKED ARRAY [saibaj..salalt] OF char);

BEGIN
  cadsal[salalt] := chr(terminador)
END; (* de cadnul *)

(* Este procedimiento escribe una cadena carácter por      *)
(* carácter en el archivo de texto a.                      *)
PROCEDURE poncad (VAR a : texto;
  cadent : PACKED ARRAY [entbaj..entalt:integer] OF char);
VAR
  i : integer;
  más : Boolean;
BEGIN
  más := true;
  i := entbaj;
  WHILE (más AND (i <= entalt)) DO

```

```

IF cadent[i] = chr(terminador)      (* ¿es el terminador? *)
THEN más := false                  (* si, no se exhibirá más *)
ELSE BEGIN
    write(a, cadent[i]);          (* exhibir un carácter *)
    i := i + 1                   (* aumentar el apuntador *)
END
END; (* poncad *)

(* loncad da como resultado la longitud de *)
(* la cadena que es su argumento *)
FUNCTION loncad
    (cadent: PACKED ARRAY [entbaj..entalt : integer] OF char) : integer;
VAR
    i : integer;
    más : Boolean;
BEGIN
    más := true;
    i : entbaj;
    WHILE (más AND (i <= entalt)) DO
        IF cadent[i] = chr(terminador)      (* ¿se halló el termino? *)
        THEN más := false                  (* si, fin de la cadena *)
        ELSE i := i + 1                   (* no, aumentar apuntador *)
    loncad := i - entbaj              (* calcular longitud cadena *)
END;

(* concad (a,b) anexa la cadena b al extremo de la cadena a *)
PROCEDURE concad (VAR a: PACKED ARRAY [bajoa..altoa: integer] OF char;
                  b: PACKED ARRAY [bajo..altob : integer] OF char);
VAR
    i, j: integer;
    más : Boolean;
BEGIN
    i := loncad(a) + 1; (* i es índice del primer carácter libre en a *)
    j := bajot           (* j es índice del primer carácter de b *)
    más := true;
    (* Mientras no se haya terminado de copiar b y quede espacio en *)
    (* a copiar los caracteres de b en el extremo de a. *)
    WHILE (más AND (i <= altoa) AND (j <= altob)) DO
    BEGIN
        a[i] := b[j];                (* copiar un carácter *)
        IF ord(a[i]) terminador < >      (* ¿es el fin de b? *)
        THEN BEGIN (* no, actualizar apuntadores de las cadenas *)
            i := i + 1;
            j := j + 1
        END
        ELSE más := false             (* si, ya se copió el terminador *)
    END;
    IF más                      (* no se ha copiado el terminador *)
    THEN IF i <= altoa

```

```

THEN a[i] := chr(terminador)          (* no se trunca *)
ELSE a[altoa] := chr (terminador)      (* se trunca *)
END; (* concat *)

(* Cadcomp (a,b) da como resultado -1, cero o +1 si a < b, a = b *)
(* o a > b, respectivamente. La más larga de dos cadenas de *)
(* longitud diferente siempre es mayor. *)
FUNCTION cadcomp (a: PACKED ARRAY [bajoa..altoa: integer] OF
                  char; b: PACKED ARRAY [bajob..altob: integer] OF
                  char): integer;
VAR
  i, j, resul: integer;
BEGIN
  i := bajoa;
  j := bajob;
  resul := 99; (*el resultado será 99 hasta detectar desigualdad *)
  WHILE (resul = 99) DO    (* mientras no se halle desigualdad *)
  BEGIN
    IF a[i] = b[j]      (* ¿son iguales los caracteres? *)
    THEN IF (a[i] = chr (terminador))  (* si, fin de cadenas? *)
         OR (i = altoa) AND (j = altob)
    THEN resul := 0   (* fin de ambas cadenas *)
    ELSE IF i = altoa (* no es fin de ambas cadenas *)
           THEN resul := -1 (* fin de a, pero no de b *)
    ELSE IF j = altob
           THEN resul := 1 (* fin de b, pero no de a *)
    ELSE BEGIN      (* ninguna cadena termina *)
      i := i + 1;  (* ambas tienen más *)
      j := j + 1
    END
    ELSE IF a[j]          (* cadenas no iguales*)
    THEN resul := -1
    ELSE resul := 1
  END;
  cadcomp := resul
END; (* cadcomp *)

(* Subcad (a, b, i, n) asigna a la cadena "a" "n" caracteres *)
(* de la cadena "b", a partir del carácter número "i" de "b". *)
(* Si no es posible almacenar "n" caracteres en "a", se *)
(* truncará el excedente. Si "i" queda fuera de la escala de *)
(* uno a loncad (b), o "n" es menor que uno, se almacenará la *)
(* cadena nula en "a". Por último, si no se pueden extraer *)
(* "n" caracteres de "b", se almacenará el número máximo que *)
(* se pueda extraer (a menos que se trunque debido a la *)
(* longitud de "a"). *)
PROCEDURE subcad (VAR a: PACKED ARRAY [bajoa..altoa: integer] OF char;
                  b: PACKED ARRAY [bajoa..altoa: integer] OF char);

```

```

    i, n : integer);
VAR
    j, lb : integer;
BEGIN
    lb := loncad (b);           (* validar i y n, y probar si la *)
                                (* cadena resultante es muy pequeña *)
    IF (i < 1) OR (i > lb)      (* i fuera de escala permitida *)
        OR (n < 1)              (* n < 1 *)
        OR (altoa = bajoa)       (* en a sólo cabe el terminador *)
    THEN nulcad (a)             (* resultado de a para estos casos *)
    ELSE BEGIN
        IF ib < i + n - 1      (* imposible obtener n caracteres *)
        THEN n := ib - i + 1;     (* ajustar n *)
        IF n > altoa - bajoa;
        THEN n = altoa - bajoa;
        j := bajoa;             (* j indica dónde guardar sig. carácter *)
        WHILE n > 0 DO(* si hay más caracteres en subcadena *)
        BEGIN
            a[j] := b[j];         (* copiar un carácter *)
            i := i + 1;           (* aumentar apunadores *)
            j := j + 1;
            n := n - 1           (* reducir número que va a copiar *)
        END;
        a[i] := chr (terminador) (* poner terminador en a *)
    END
END;
(* índice (a,b) produce como resultado un entero que representa      *)
(* la primera posición de la izquierda de la cadena b en la          *)
(* cadena a. Si b no aparece como subcadena en a, el resultado      *)
(* de la función será cero.                                         *)
FUNCTION índice (
    a: PACKED ARRAY [bajoa..altoa: integer] OF char;
    b: PACKED ARRAY [bajob..altob: integer] OF char): integer;
VAR
    i,j,k,la,lb: integer;
    está: Boolean;
BEGIN
    la := loncad(a);
    lb := loncad(b);
    IF lb > la                (* si b es más larga que a *)
    THEN índice := 0             (* es imposible que esté en a *)
    ELSE BEGIN
        i := 1;                  (* comenzar por el primer carácter de a *)
        está := false;
        (* seguir hasta encontrarla o que ya no pueda caber *)
        WHILE (NOT está AND (i <= la - lb + 1)) DO
        BEGIN
            j := bajoa + i - 1;

```

```

k := bajob;
está := true; (* se supone que se encontrará aquí *)
(* probar todos los caracteres de b con la subcad de a en i *)
WHILE (está AND (k <= bajob + lb - 1)) DO
  IF a[j] < > b[k]          (* diferentes, no es subcad. *)
  THEN está := false
  ELSE BEGIN (* concuerdan, pasar a sig. caracteres *)
    j := j + 1;
    k := k + 1
  END;
  IF NOT está
  THEN i := i + 1
END;           (* si se falló, pasar a la sig. subcad *)
IF found      (* si hubo éxito, el resultado es el índice *)
THEN índice := i
ELSE índice := 0      (* si no, el resultado es cero *)
END
END; (* índice *)

```

(\* verif (a,b) da como resultado el índice del primer carácter \*)  
 (\* de a que no está en b. Si todos los caracteres de a aparecen \*)  
 (\* también en b, el resultado de verif es cero. \*)

```

FUNCTION verif (a: PACKED ARRAY [bajoa..altoa: integer] OF char;
               b: PACKED ARRAY [bajob..altob: integer] OF char): integer;
VAR
  i, la : integer;
  c : PACKED ARRAY [1..2] OF char;
  más := Boolean
BEGIN
  la := loncad(a);
  IF loncad(b) = 0                      (* casos especiales *)
  THEN IF la = 0
    THEN verif := 0 (* si b y a son nulas, el resultado es cero *)
    ELSE verif := 1 (* primer carácter de a que no está en b *)
  ELSE BEGIN
    c[2] := chr (terminador); (* c es cadena de un carácter *)
    i := 1;
    más := true;
    WHILE ((i <= la) AND más) DO
    BEGIN
      c[1] := a[bajoa + i - 1];        (* obtener un carácter *)
      IF índice (b,c)                (* ¿está c en b? *)
      THEN i := + 1                  (* se sigue buscando *)
      ELSE más := false             (* no, se puede terminar *)
    END;
    IF más      (* si todos los caracteres de b estaban en a *)
    THEN verif := 0
  END;

```

```

ELSE verif := i (* si no, el resultado es el índice de
la divergencia *)  

END  

END (* de verif *)  

(* Leelin captura la siguiente línea del archivo de texto f y lo      *)  

(* asigna a la cadena a. Si no cabe la línea completa, se trunca.      *)  

(* Si se encuentra un fin de archivo antes de un fin de línea, la      *)  

(* función deja la parte de la linea que precede al eof en a, y      *)  

(* da como resultado false. En caso contrario el resultado es true      *)  

FUNCTION leelin (VAR f: text;  

                  VAR a: PACKED ARRAY [bajoa..altao: integer] OF char): Boolean;  

VAR  

  c : char;  

  i : integer;  

BEGIN  

  i := bajoa;  

  WHILE NOT (eof(f) OR eoln(f)) DO      (* si no se ha terminado *)  

    BEGIN  

      read (f,c);  

      IF i < altao          (* si queda espacio en la cadena *)  

        THEN BEGIN  

          a[i] := c;      (* introducir el nuevo carácter *)  

          i := i + 1      (* actualizar long. de la cadena *)  

        END  

      END;  

      a[i] := chr (terminador);      (* terminar la cadena *)  

      IF eol..(f)  

        THEN BEGIN  

          readln(f);      (* pasar por alto el fin de línea *)  

          leelin := true  (* y dar como resultado true *)
        END
      ELSE leelin := false      (* en caso contrario, resultado es false *)
    END; (* de leelin *)
  (* sigpal obtiene la siguiente palabra del texto y da como      *)  

  (* resultado true, o resulta false si terminó el archivo      *)  

FUNCTION sigpal (VAR lapal : palabra) : Boolean;  

VAR  

  más : Boolean;  

  i : integer;  

  línea2 : PACKED ARRAY [1..lonmáxlin] OF char;  

BEGIN  

  más := true;  

  WHILE y AND (verif línea, espacio en blanco) = o DO  

    BEGIN  

      más := leelin (input, línea);  (* capturar siguiente línea *)  

      IF más                      (* si no ha terminado el archivo *)
        THEN BEGIN

```

(\* cambiar caracteres de tabulación por espacios en blanco \*)

```

FOR i := 1 TO loncad(línea) DO
  IF línea[i] = chr(9)
    THEN línea[i] := ' ';
    concad (línea, blanco)
  END
END;
IF más
THEN BEGIN
  i := verif (línea, blanco); (* encontrar primer carácter que
                                no es espacio en blanco *)
  (* almacenar línea sin espacios a la izq. en linea2 *)
  subcad (línea2, línea, i, loncad(línea)-i + 1);
  i := índice (línea2, blanco); (* encontrar primer espacio *)
  (* almacenar la palabra (todos los caracteres antes del espacio *)
  subcad (lapal, línea2, 1, i-1)
  (* almacenar línea2 sin la primera palabra en línea *)
  subcad (línea, línea2, i, loncad(línea2)-i + 1)
  END;
  sigpal := más
END; (* de sigpal *)

PROCEDURE iniciales;
BEGIN
  nulcad(línea);          (* no se ha capturado línea de texto *)
  númpal := 0;             (* no hay palabras en la tabla *)
  blanco[1] := ' ';        (* blanco es cadena de un espacio en blanco *)
  blanco[2] := chr(terminador);
END;
(* Actuliza busca una palabra en la tabla y le suma uno para      *)
(* actualizar su cuenta. Si la palabra no está en la tabla y      *)
(* hay espacio, se agrega la palabra nueva al final y se le      *)
(* asigna uno a su cuenta. Si no hay espacio, se "pierde" una      *)
(* palabra nueva                                              *)
PROCEDURE actualiza (nuevapal: palabra);
VAR
  está, más: Boolean;
  i : 0..máxpal;
BEGIN
  está := false;
  i := 1;
  más := i = <= númpal; (* más es false si la tabla está vacía *)
  WHILE NOT está AND más DO (* si se debe seguir buscando *)
    BEGIN
      IF compcad(nuevapal, palabras[i] = 0 (* ¿es esta palabra? *)
      THEN BEGIN
        está := true;           (* sí, se puede terminar *)
      END;
    END;
  END;
END;

```

```

        cuentas[i] := cuentas[i] + 1 (* actualizar cuenta *)
    END
ELSE IF i = númpal      (* no se ha encontrado todavía *)
    THEN i := i + 1          (* pero se debe seguir
                                buscando *)
    ELSE más := false       (* terminó la búsqueda *)
END;
IF NOT está AND (númpal = máxp)  (* si no se halló, y hay *)
THEN BEGIN                   (* espacio para una palabra nueva *)
    númpal := númpal + 1;    (* aumentar cuenta de *)
                                palabras *)
    copcad (palabras[númpal], nuevapal); (* agregar la palabra
                                              nueva *)
    cuentas[númpal] := 1      (* ya se usó una vez *)
END
END; (* se actualiza *)

```

(\* clasif hace una clasificación por inserción de palabras y cuentas \*)

```

PROCEDURE clasif;
VAR
    i,j, máx : integer;
    temp : palabra;
BEGIN
    FOR i := númpal DOWNTO 2 DO
    BEGIN
        máx := 1;
        FOR j := 2 TO i DO
            IF compcad (palabras[máx], palabras[j]) = -1
            THEN máx := j;
            copcad (temp, palabras[i]);
            copcad (palabras[i], palabras[máx]);
            copcad (palabras[máx], temp);
            j := cuentas[i];
            cuentas[j] := cuentas[máx];
            cuentas[máx] := j
    END
END; (* clasif *)

```

(\* Databla exhibe la tabla de palabras y cuentas. Los encabezados \*)
(\* se ajustan automáticamente para las diferentes longitudes \*)
(\* máximas de las palabras \*)

```

PROCEDURE databla;
VAR
    i,j,n: integer;
BEGIN
    (* Exhibir los encabezados de la tabla. *)

```

(\* Calcular númer de espacios que conviene \*)  
colocar en ambos lados de una \*)

(\* "palabra" para centrarla en columnas de ancho lonmáxpal. \*)

```

n := (lonmáxpal -7 (* loncad ('Palabra') *)) DIV 2;
FOR i := 1 TO N DO write ('');
write ('Palabra');
FOR i := n + 8 TO lonmáxpal DO write (' ');
writeln ('Cuenta');
FOR i := 1 TO lonmáxpal DO write ('—');
writeln ('-----');
FOR j := 1 TO númpal DO
BEGIN
    pocad(output, palabras[j]);          (* exhibir una palabra *)
    FOR i := loncad(palabras[j]) + 1 TO lonmáxpal + 1 DO
        write ('');           (* espacios hasta la columna correcta *)
    writeln (cuentras[j]:6)            (* exhibir la cuenta *)
END
END; (* de databla *)

```

(\* Programa principal \*)

```

BEGIN
    iniciales;                      (* preparar procesamiento *)
    WHILE sigpal (unapal) DO actualiza (unapal); (* contar palabras *)
    clasif;                         (* clasificar palabras en orden ascendente *)
    databla;                        (* exhibir las palabras y veces que aparecen *)
END.

```

---

## SECCIÓN 10.5 TÉCNICAS DE PRUEBA Y DEPURACIÓN

Uno de los problemas que se presentan con mayor frecuencia al manejar cadenas de caracteres en Pascal es no darse cuenta de que las constantes de cadena de caracteres representan en realidad arreglos empacados de caracteres cuyo subíndice tiene un límite inferior de uno. Esto implica que tales constantes no se pueden asignar a arreglos no empacados o arreglos empacados cuyos subíndices tengan un límite inferior diferente de uno. Además, el límite superior del subíndice de la variable de arreglo empacado que se usa en el lado izquierdo de una proposición de asignación debe ser exactamente el mismo que el número de caracteres de la constante o variable de cadena que aparece en el lado derecho. Por ejemplo, si se hacen las declaraciones

VAR

```

cadena10 :PACKED ARRAY [1..10] OF char;
cadena10x : PACKED ARRAY [0..9] OF char;

```

entonces las proposiciones de asignación

**cadena10 := '0123456789'**

y

**cadena10 := 'Ecuador '**

son válidas, pero

**cadena10 := 'Antofagasta'**

**cadena10 := 'París'**

y

**cadena10x := 'Inglaterra'**

no lo son. Las mismas reglas se aplican a las comparaciones de cadenas. Aunque se puede escribir

**'Pascal' > 'Modula'**

no se permite escribir

**'Peras' < > 'Manzanas'**

porque las cadenas no tienen la misma longitud. Todos estos problemas se eliminan al realizar las operaciones mediante los procedimientos y funciones que se presentaron en la última sección.

La diferencia entre las letras mayúsculas y minúsculas puede provocar también confusiones. Dado que estos caracteres tienen valores ordinales diferentes, no son iguales. Por ejemplo, si se comparan las constantes de cadena 'ALTAS' y 'altas' se verá que no son iguales y, lo que es peor, la comparación puede producir resultados diferentes en distintas máquinas. Es decir, muchas máquinas pueden indicar que 'ALTAS' es más pequeño que 'altas', pero en otras máquinas 'ALTAS' es mayor que 'altas'. Si se escribe código con la intención de que sea transportable, se debe prestar atención especial a este problema para garantizar que el código que incluya comparaciones de caracteres no produzca resultados inesperados.

Si se usan cadenas de caracteres de longitud fija para almacenar datos que no son forzosamente tan largos como la cadena de longitud fija, es conveniente asignar un valor constante inicial (normalmente espacios en blanco) a todos los caracteres no utilizados de la cadena. Esto se puede realizar de dos maneras. En primer lugar, la cadena se puede llenar completamente de espacios en blanco antes de almacenar los datos de caracteres en ella. Ésta es la más sencilla de las dos técnicas, pero es más costosa, ya que se hace referencia dos veces a algunos caracteres de la cadena, una vez para almacenar un espacio en blanco y otra para almacenar los datos reales. La segunda técnica consiste en llenar parte de la cadena con los datos reales y más tarde almacenar espacios en la porción que no se usó. Esto es más

eficiente pero también más complicado, ya que se debe utilizar un ciclo para almacenar los espacios. Ambas técnicas se ilustraron en los programas que se mostraron en este capítulo.

Se mencionará un último problema que tiene que ver con el número de caracteres que se permiten en las variables de cadena. Supóngase, por ejemplo, que se está escribiendo un programa que maneja nombres de personas. Es posible prever que el apellido más largo de un individuo puede tener 15 caracteres, pero muchas personas tienen apellidos más largos. Si no se verifican los límites de arreglos al almacenar caracteres en una cadena, es posible que se presente un error de subíndice. Por otro lado, si se usan procedimientos y funciones similares a los que se presentaron en la sección 10.4, es posible perder caracteres por el truncado de nombres más largos. Truncar no siempre es un error, pero si existe la posibilidad de ello es conveniente que el código incluya pruebas específicas para detectar esta situación y emprender acciones correctivas.

He aquí una lista de recordatorios importantes que se refieren al uso de cadenas en Pascal.

## RECORDATORIOS DE PASCAL

- Los arreglos empacados se declaran mediante las palabras reservadas PACKED ARRAY.
- Los componentes de un arreglo empacado no se pueden pasar como parámetros a una función o procedimiento.
- Las variables de cadena se declaran así:

PACKED ARRAY [1..longitud máxima] OF char;

- El límite inferior del subíndice de las variables de cadena debe ser uno.
- Los procedimientos estándar *pack* y *unpack* de Pascal sirven para hacer la conversión entre arreglos empacados y desempacados.
  - Las constantes y variables de cadena del mismo tamaño y tipo pueden asignarse directamente y compararse mediante los operadores relacionales.
  - Los parámetros conformantes de arreglo sirven para pasar parámetros de arreglo de longitud variable.
  - Los componentes de un parámetro de arreglo conformante pueden ser de cualquier tipo, incluso otro arreglo conformante.

## SECCIÓN 10.6 REPASO DEL CAPÍTULO

En este capítulo se analizaron las cadenas de caracteres y las variables de cadena que se declaran como arreglos empacados de caracteres. Las cadenas se pueden exhibir o imprimir en su totalidad mediante una sola proposición *write*, y no es necesario exhibir la cadena carácter por carácter. Sin embargo, para capturar las cadenas como datos de entrada es preciso introducirlas carácter por carácter. Las cadenas se pueden comparar mediante los operadores relacionales. También se analizaron dos procedimientos estándar del Pascal, *pack* y *unpack*, que pueden utilizarse pa-

ra convertir arreglos completos del formato no empacado en empacado, y viceversa.

En el capítulo se incluyó asimismo la aplicación de las cadenas a diversos problemas. Se escribieron y analizaron varias rutinas de procesamiento de cadenas, entre ellas la concatenación. Como ejemplo de resolución de problemas mediante cadenas se escribió un editor de textos sencillo. Se analizaron los parámetros conformantes de arreglo y se aplicaron a varias rutinas de manipulación de cadenas. Estas rutinas se usaron más tarde para resolver un problema de procesamiento de cadenas relativamente extenso.

En seguida se presenta un resumen de las características de Pascal que se analizaron en este capítulo y que puede servir como referencia en el futuro.

## REFERENCIAS DE PASCAL

- 1 Arreglo empacado: arreglo cuyos componentes se pueden empacar en forma más compacta que lo normal para ahorrar memoria, aunque posiblemente a expensas de la rapidez de procesamiento.

Ejemplo de declaración:

VAR

cadena: PACKED ARRAY [1..10] OF char;

- 2 Variables de cadena: variables de arreglo empacado.

- Se pueden exhibir mediante una sola proposición *write* o *writeln*:

*writeln (cadena);*

- Se pueden comparar con otras cadenas o constantes de cadena mediante los operadores relationales usuales:

*cadena > 'California'*

- Se les puede asignar el valor de una constante o variable de cadena de la misma longitud:

*cadena := 'Washington'*

- 3 Procedimientos estándar de Pascal:

- “*pack (arreglon, inicio, arregloe)*” copia los caracteres necesarios a partir del elemento *arreglon[inicio]* de un arreglo no empacado para llenar el arreglo empacado *arregloe*.
- “*unpack (arregloe, arreglon, inicio)*” copia todos los caracteres del arreglo empacado *arregloe* en el arreglo no empacado *arreglon* a partir de *arreglon[inicio]*.

- 4 Parámetros conformantes de arreglo: permiten emplear arreglos con diferentes límites para los subíndices como parámetros de un solo procedimiento o función, ya que proporcionan a la función los límites inferior y superior del subíndice del parámetro de arreglo. Ejemplo:

```
PROCEDURE loncad (VAR c:  
PACKED ARRAY [bajo..alto : integer] OF char);
```

## Avance del capítulo 11

En el siguiente capítulo se continuará el estudio de los tipos de datos estructurados con el análisis de los registros. A diferencia de los arreglos, los componentes de un registro pueden tener diferentes tipos de datos. Los arreglos permiten procesar elementos de tipo similar en forma de grupo, pero en ocasiones se desea procesar datos de diferentes tipos en forma de grupo. Como verá el lector, el tipo de datos de registro se puede utilizar de manera efectiva en muchos problemas, como en el de un sistema de expedientes de estudiantes.

## Palabras clave del capítulo 10

arreglo empacado  
 cadena  
 cadena de caracteres  
 concatenación  
 empacar  
 desempacar  
 pack  
 palabra  
 parámetro conformante de arreglo  
 unpack  
 variable de cadena

## EJERCICIOS DEL CAPÍTULO 10

### ★ EJERCICIOS ESENCIALES

- 1 ¿Es aceptable el siguiente segmento en Pascal estándar? De ser así, ¿cuál será el resultado de su ejecución? Si no se permite, ¿por qué? Supóngase que *c* se declara como variable de caracteres y *n* como entera.

```
(n := 0;  
FOR c := 'A' TO 'Z' DO n := n + 1
```

- 2 Determínese si las siguientes expresiones son validas o no. En el caso de las expresiones válidas, determínese el tipo y valor de su resultado.

- a)  $\text{ord}(\text{'A'}) = \text{ord}(\text{'B'})$
- b)  $\text{chr}(-3) + 4$
- c) ‘Pérez’ = ‘PÉREZ’
- d) ‘A’ = ‘A’
- e)  $\text{ord}(\text{'Sin costo'})$
- f) ‘MENSAJE’[4]
- g)  $\text{ord}(\text{succ}(\text{chr}(9)))$

- 3 El lector probablemente ya sabe que una cadena como ‘MEMORIA’ tiene en realidad el mismo tipo de datos que

**PACKED ARRAY [1..7] OF char;**

por lo que es posible asignar esa constante a una variable de este tipo. Por tanto, el segmento

```
VAR s7: PACKED ARRAY [1..7] OF char;
s7 := 'MEMORIA';
```

es completamente legal.

¿Por qué, entonces, no se permite el siguiente código?

```
VAR s1: PACKED ARRAY [1..1] OF char;
s1 := '';
```

- 4 Muchos sistemas de cómputo almacenan los arreglos de caracteres en la forma más compacta posible sin que se especifique explícitamente la palabra clave **PACKED**. Si se usa un sistema de cómputo de este tipo, ¿qué caso tiene emplear definiciones de **PACKED ARRAY**?

- 5 Si se suponen las declaraciones

```
VAR
  cadena: PACKED ARRAY [1..10] OF char;
  c : char;
  i : integer;
```

¿qué salida producirá el siguiente segmento de programa?

```
cadema := '0123456789';
FOR i := 1 TO 10 DO
BEGIN
  c := cadena[i];
  IF NOT odd(ord(c) — ord('O'))
    THEN cadena[i] = ''
END;
writeln (cadena: 10)
```

## ★ ★ EJERCICIOS IMPORTANTES

- 6 En Pascal se pueden utilizar apóstrofos para encerrar uno o más caracteres, pero no cero caracteres, como sería ‘ ’. Algunos lenguajes sí permiten esto y lo manejan correctamente como una cadena de caracteres nula. ¿Por qué piensa el lector que el Pascal estándar no permite esta constante?
- 7 Supóngase que una cadena contiene cuatro caracteres que representan una hora del día (es decir, entre ‘0000’ y ‘2359’). Escríbase un segmento en Pascal que cree una cadena de ocho caracteres que represente el tiempo correspondiente en un reloj de 12 horas con un signo de dos puntos para separar las horas y minutos, un espacio en blanco y las letras correspondientes AM o PM. Por ejemplo, la cadena ‘1200’ daría como resultado ‘12:00 PM’ y ‘0159’ daría ‘01:59 AM’.

## ★ ★ ★ EJERCICIOS ESTIMULANTES

- 8 Escríbase un procedimiento llamado *poncoma* que tenga dos parámetros. El primero es un entero en la escala de uno a 999999. El segundo parámetro (de salida) debe ser la representación en caracteres del entero, justificada a la derecha, con una coma en la posición apropiada si el valor requiere más de tres dígitos. Por ejemplo, si el entero es 999, la cadena de caracteres resultante debe ser ‘ 999’. Si el entero es 19999, la cadena deberá ser ‘ 19,999’.
- 9 El Pascal estándar no permite que aparezcan variables booleanas como parámetros en invocaciones de los procedimientos *read* o *readln*. Escríbase una función booleana llamada *leebool* que pasa por alto espacios en blanco a la izquierda y caracteres de tabulación y fin de linea, y dé como resultado *true* si el siguiente carácter es ‘T’ o ‘t’ y *false* en caso contrario.

**PROBLEMAS DEL CAPÍTULO 10 PARA RESOLUCIÓN EN COMPUTADORA**

## ★ PROBLEMAS ESENCIALES

- 1 Escríbase un programa en Pascal que transforme un archivo de texto de entrada, el cual contiene caracteres de tabulación, de manera que éstos se sustituyan por el número apropiado de espacios en blanco. Cada carácter de tabulación debe sustituirse por el número de espacios en blanco (por lo menos uno pero no más de ocho) necesarios para que el número total de caracteres de la línea de salida sea un múltiplo de ocho en ese momento.

Ejemplo de entrada (los caracteres de tabulación se representan mediante ^|):

Esta^|es^|una^|prueba. <eoln>  
^|^|También ésta.^| <eoln>

Ejemplo de salida:

Ésta es una prueba.<eoln>  
También ésta. <eoln>

- 2 El programa realiza la operación inversa a la que lleva a cabo el programa del problema 1. Sustitúyanse en la medida de lo posible los espacios en blanco de los datos de entrada por caracteres de tabulación.

Ejemplo de entrada:

Otra personalidad extraña

Ejemplo de salida:

^| Otra^| personalidad^| extraña

- 3 Escríbase un programa para centrar y subrayar un título que se lee de la primera línea de datos de entrada. Para centrar el título se puede determinar primero el número de caracteres que tiene el título, restárselos a la anchura de la línea de salida y exhibir la mitad de ese número de espacios en blanco seguidos por el título. El subrayado se logra mediante la impresión de un signo de menos debajo de todos los caracteres del título que no sean espacios en blanco.

Ejemplo de entrada:

Despertar a los muertos

Ejemplo de salida (supóngase que la línea tiene 80 caracteres):

Despertar a los muertos

- 4 Escríbase un programa para convertir cualquier dígito que se presente en los datos de entrada, en su equivalente en español, sin modificar los demás caracteres.

Ejemplo de entrada

Se dieron 4 valores, a saber 7, 9,  
3 y 0. El resultado tuvo 10 dígitos.

Ejemplo de salida:

Se dieron cuatro valores, a saber siete, nueve  
tres y cero. El resultado tuvo unocero dígitos.

## ★★ PROBLEMAS IMPORTANTES

- 5 Cada una de las cinco líneas de datos de entrada contiene una representación de una carta de la baraja en forma de dos cadenas separadas por lo menos por

un espacio en blanco y posiblemente precedidas o seguidas por espacios adicionales. Determinese el valor que tendría la mano en un juego de póquer.

Ejemplo de entrada:

```
9 ESPADA <eoln>
8      DIAMANTE <eoln>
REINA TRÉBOL <eoln>
      10 TRÉBOL      <eoln >
JACK CORAZÓN
```

Ejemplo de salida:

**CORRIDA**

- 6 Cada una de las líneas de un archivo de entrada contiene un nombre de la forma apellido, coma y nombre de pila. No se incluirán espacios dentro del apellido o el nombre, pero pueden aparecer en cualquier otro lugar. Escríbese un programa en Pascal que lea los nombres, los clasifique según el apellido y el nombre y los exhiba comenzando con el nombre de pila.

Ejemplo de entrada:

```
Chávez, Federico
          Blanes, Hugo
Flores , Susana
          Vázquez , Noé
Pérez, Beatriz
```

Ejemplo de salida:

```
Hugo Blanes
Federico Chávez
Susana Flores
Beatriz Pérez
Noé Vázquez
```

### ★ ★ ★ PROBLEMAS ESTIMULANTES

- 7 Un profesor quiere un programa para calificar exámenes de opción múltiple. Se proporciona un archivo de entrada que tiene la clave de respuestas y las contestaciones de cada estudiante. Cada una de las líneas tiene información para identificación del estudiante en las primeras nueve columnas, seguida de un número variable de respuestas. El número de identificación del estudiante de la primera línea se debe pasar por alto, ya que las respuestas representan las contestaciones correctas a las preguntas. Las respuestas son caracteres alfabéticos de mayúsculas o espacios en blanco. Los espacios indican que esa respuesta no se

debe calificar. Las líneas restantes de los datos de entrada representan contestaciones de los estudiantes. Se deben pasar por alto las respuestas de los estudiantes a preguntas que no se indican en la clave de respuestas. Escríbase el programa que califique los exámenes y proporcione la identificación del estudiante, el número de respuestas correctas y la calificación como porcentaje de cada estudiante.

Ejemplo de entrada:

```
ADABECC DBE <eoln>
808732533ADABECCDDBEF <eoln>
808749266ABADEFBDABEC <eoln>
813555121ADBCECCDDCAD <eoln>
```

Ejemplo de salida:

Estudiante	Correctas	Porcentaje
808732533	10	100
808749266	5	50
8113555121	6	60

- 8 Escríbase un programa para indizar todas las declaraciones de variables en un programa en Pascal. Supóngase que las declaraciones de variables son de una forma limitada. El ejemplo de datos de entrada muestra las formas de declaración válidas. La salida debe incluir todos los nombres de variables y el número de línea del programa donde se encontró la declaración. De manera opcional, clasifíquese la lista por nombre de variable e inclúyase el tipo de cada variable.

Ejemplo de entrada:

```
VAR v1, v2: integer;
VAR
  v5 , v6 :
    real;
VAR v3,v4 ,v7:char;
```

Ejemplo de salida (incluyendo el tipo):

v1	entera	1
v2	entera	1
v3	de carácter	5
v4	de carácter	5
v5	real	3
v6	real	3
v7	de carácter	5



# CAPÍTULO 11

## CAPÍTULO 11 REGISTROS



# REGISTROS

Después de completar este capítulo, el lector deberá ser capaz de

- Reconocer y aplicar el tipo de datos estructurado de registro
- Declarar y aplicar registros jerárquicos (registros anidados)
- Declarar y aplicar arreglos de registros
- Reconocer y aplicar la proposición WITH
- Clasificar un arreglo de registros
- Reconocer y aplicar registros variantes (opcional)
- Resolver, probar y depurar problemas que usen registros

## PANORAMA GENERAL DEL CAPÍTULO

El estudio de los tipos de datos estructurados continuará con la introducción de **registros**. Los registros son similares a los arreglos en tanto que están formados por componentes. Pero, a diferencia de los arreglos, los componentes de los registros pueden ser de diferentes tipos de datos. Por ejemplo, el registro de los antecedentes de un estudiante puede incluir su nombre y dirección (tipo de datos de arreglo empacado), edad (entero), promedio de calificaciones (número real) y grado: primer año, segundo año, tercer año o cuarto año (tipo enumerado.)

En la sección 11.1 se mostrará la forma de construir un registro mediante una definición de tipo y una declaración de variable. Se describirán los métodos empleados para tener acceso a los diferentes componentes de un registro. Los registros pueden contener arreglos como componentes, así como otros registros. Pueden anidarse, y en la sección 11.2 se analizan los registros anidados. Recuérdese que los componentes de un arreglo pueden ser de cualquier tipo de datos, incluso tipos estructurados como son los registros. Se estudiarán de manera detallada los arreglos de registros. Hacer referencia a componentes de estructuras complejas, como son los arreglos de registros anidados, puede producir expresiones largas y tediosas. Pascal cuenta con la proposición WITH que simplifica este proceso. Se examinarán algunos ejemplos en la sección 11.2.

En la sección de resolución de problemas se estudiará el problema de clasificar un arreglo de registros. Este problema se diferencia del problema de clasificación, antes presentado, en dos aspectos: los componentes son registros en vez de enteros, y los elementos que se han de clasificar son cadenas de caracteres incluidas dentro de cada registro. También se incluye una sección opcional que trata de registros variantes. Éstos son registros que contienen una parte variante de componentes que pueden cambiar durante la ejecución del programa. La sección de técnicas de prueba y depuración incluye algunos problemas comunes relacionados con los registros.

Un arreglo consta de varios componentes, todos del mismo tipo. Sin embargo, existen muchas aplicaciones en las que los componentes de un tipo estructurado deben ser de tipos de datos diferentes. Para manejar estos casos, Pascal incluye un tipo de datos estructurado llamado *registro*. Por ejemplo, supóngase que una empresa de mercadotecnia se especializa en encuestas a consumidores. La información que normalmente requiere esta compañía es la siguiente:

Nombre de la persona (35 caracteres máximo)

Código postal (entero)

Edad (entero)

Número telefónico (12 caracteres)

Respuesta (de acuerdo o en desacuerdo)

Esta información se puede incluir en forma apropiada en un registro de Pascal mediante la siguiente definición y declaración:

#### TYPE

```
consumidor = RECORD
    nombre: PACKED ARRAY [1..35] OF char;
    códpost: 0..99999
    edad: 21..99
    teléfono: PACKED ARRAY [1..12] OF char;
    respuesta: (acuerdo, desacuerdo)
END;
```

#### VAR

```
persona: consumidor;
```

En la definición de tipo, el identificador de tipo de registro *consumidor* va seguido de un signo de igual y una lista, separada por signos de punto y coma, de descripciones de los componentes llamadas *secciones del registro*, encerrada entre las palabras reservadas RECORD y END. Cada uno de los componentes del registro tiene un identificador único llamado *identificador de campo*. La sección de variables declara a *persona* como variable de registro.

El siguiente ejemplo muestra datos que podrían estar contenidos en la variable de registro *persona*:

nombre	'FERNANDO G. HIGUERA'
códpost	10021
edad	39
teléfono	212-551-0000
respuesta	acuerdo

La forma general de una definición TYPE de registro es la siguiente:

**TYPE**

```
  identificador = RECORD
    identificador de campo : tipo;
    identificador de campo : tipo;
    ...
    identificador de campo : tipo;
  END;
```

El diagrama de sintaxis correspondiente se muestra en la figura 11-1.

He aquí otro ejemplo de definición de registro y declaración. ¿Puede el lector encontrar una aplicación para este registro?

**TYPE**

```
  info = RECORD
    nombre: PACKED ARRAY [1..35] OF char;
    grado: (primero, segundo, tercero, cuarto);
    prom: real;
    créditos, créditostransf: 0..máxint;
    sexo: (masc, fem)
  END;
```

**VAR** .

```
  estudiante: info;
```

### Denominadores de campo

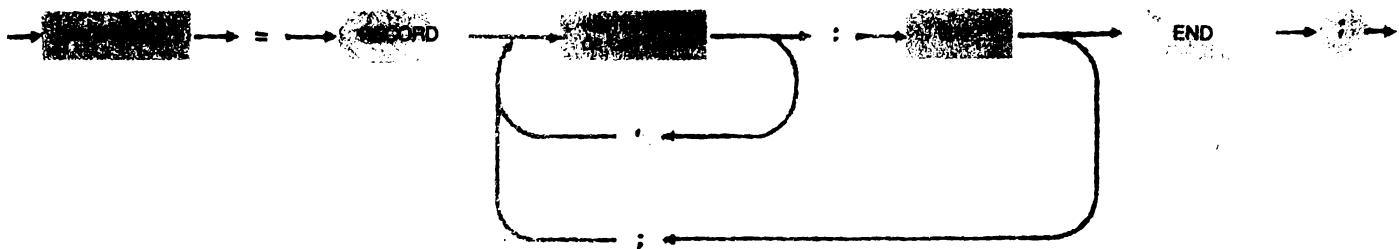
Para tener acceso a un componente de un arreglo se utilizó un subíndice o índice. Debe ser posible también tener acceso a un componente deseado de un registro. Para ver cómo se hace esto, se volverá a la variable de registro *persona*. Para tener acceso al nombre dentro del registro se emplea el identificador de la variable de registro (*persona*) seguida de un punto, y después el identificador de campo del campo del nombre (*nombre*):

*persona.nombre*

Esta expresión se llama *denominador de campo*. Para tener acceso a los demás componentes se usan los siguientes denominadores de campo:

<i>persona.códpost</i>	<i>persona.edad</i>
<i>persona.teléfono</i>	<i>persona.respuesta</i>

Los denominadores de campo de tipos simples (no estructurados) se manejan igual que las variables ordinarias de tipos simples. Por ejemplo, se pueden emplear en expresiones y proposiciones de asignación. Las siguientes proposiciones de asignación asignan valores a todos los denominadores de campo de la variable de registro *persona*:



**Figura 11-1** Diagrama de sintaxis de los registros.

```

persona.nombre := 'Charles Dickens
persona.códpost := 00001;
persona.edad := 55;
persona.teléfono := '213-861-4134';
persona.respuesta:= desacuerdo;

```

Para leer y exhibir los valores de una variable de registro se usan los denominadores de campo y las reglas que se aplican al tipo de datos correspondiente. Por ejemplo, la proposición

```
writeln (persona.nombre)
```

exhibe el nombre que está en el registro, ya que el tipo que corresponde a *persona.nombre* es el de arreglo empacado de caracteres. Si *persona.nombre* no estuviera empacado sería necesario exhibir cada carácter por separado, ya que no es posible especificar arreglos completos en la lista de parámetros de las proposiciones *writeln*. El valor del denominador de campo *persona.respuesta* es de un tipo enumerado, por lo que no es posible leer o exhibir ese valor directamente. En vez de ello se utiliza una proposición CASE para indicar el valor de este componente:

```

CASE persona.respuesta OF
    acuerdo: writeln ('El consumidor está de acuerdo');
    desacuerdo: writeln ('El consumidor no está de acuerdo');
END;

```

He aquí otro ejemplo que emplea denominadores de campo con la variable de registro *persona* para determinar si la persona pasa de los 65 años.

```

IF persona.edad > 65
THEN writeln ('La persona tiene más de 65 años.')

```

Éstos son algunos otros puntos que conviene recordar en lo que toca a los registros:

- Los registros se pueden pasar como parámetros a funciones y procedimientos.
- El resultado de una función no puede ser un registro.

- Los registros se pueden asignar a registros de tipo idéntico.  
Supóngase que se tienen esta definición y declaraciones:

**TYPE**

**antecedentes = RECORD**

**nombre: PACKED ARRAY [1..35] OF char;**

**edad: 1..99**

**categoría: (ciudadano, residente, visitante);**

**sexo: (masc, fem)**

**END;**

**VAR**

**persona1, persona2 : antecedentes;**

Puesto que *personal* y *persona2* tienen tipos idénticos, la proposición de asignación

**persona2 := persona1**

está permitida y ocasiona la asignación de los valores de cada uno de los denominadores de campo de *personal* a los denominadores de campo respectivos de *persona2*.

- El tipo que se asocia a cualquier identificador de campo de un registro puede ser otro registro. Esta idea, los registros anidados o jerárquicos, se estudiará en la siguiente sección.

Esta sección termina con un ejemplo de un procedimiento que lee un registro completo de un archivo de texto.

### Problema 11.1

*Escríbase un procedimiento para leer un valor para cada campo de un registro que se define así:*

**TYPE**

**resultado = RECORD**

**apellido: PACKED ARRAY [1..20] OF char;**

**notas: ARRAY [1..5] OF integer;**

**letraid: 'A'..'Z';**

**calif: (excelente, bien, regular, mal)**

**END;**

*La línea*

González 90 80 95 92 91 Z A

*contiene datos de entrada representativos, donde el código para excelente es A, bien B, regular C y mal D.*

```
PROCEDURE leer registro (VAR datos: resultado);
(* Leer un registro de tipo resultado del archivo de entrada *)
VAR
    índice: integer;
    uncar: char;
BEGIN
    FOR índice := 1 TO 20 DO          (* capturar apellido *)
        BEGIN
            read (uncar);
            datos.apellido[índice] := uncar
        END;
    FOR índice := 1 TO 5 DO          (* capturar notas de examen *)
        read (datos.notas[índice]);
    read (uncar);                  (* pasar por alto espacios *)
    WHILE uncar = ' ' DO           (* antes de letraid *)
        read (uncar);
        datos.letraid := uncar;
    read (uncar);                  (* asignar letraid *)
    read (uncar);                  (* pasar por alto espacios *)
    WHILE uncar = ' ' DO
        read (uncar);
    IF uncar IN ['A', 'B', 'C', 'D']      (* validar código de calif *)
        THEN CASE uncar OF             (* convertir en tipo enumerado *)
            'A': data.calif := excelente;
            'B': data.calif := bien;
            'C': data.calif := regular;
            'D': data.calif := mal
        END
    ELSE writeln ('Calificación no válida.') (* diagnosticar error *)
END;
```

## EJERCICIOS DE LA SECCIÓN 11.1

- 1 Escribáse una definición de registro para los siguientes datos:
  - a) Un registro bancario que consta de nombre y dirección de la cuenta, número de seguro social, saldo actual e intereses acumulados.
  - b) Un directorio telefónico que incluya nombre, dirección y número de teléfono.
  - c) Un expediente criminal con nombre verdadero, alias, edad, altura, peso, color de los ojos y número de arrestos previos.
- 2 Determinése cuáles de las siguientes definiciones de registro son válidas.
  - a) TYPE  
estado = RECORD  
 nombre: PACKED ARRAY [1..30] OF char;  
 edad: integer;  
 sexo: (masc, fem)

b) TYPE  
ejemplo = RECORD  
prueba: 0..100;  
final: 'A'..'F';  
orden: 1..100  
END;

c) TYPE  
franco = RECORD  
prueba: 0..100;  
examen: 0..10;  
prueba: 0..100  
END;

d) TYPE  
califs = RECORD  
nombre,  
dirección: PACKED ARRAY [1..30] OF char;  
prueba,  
examen: 0..100;  
tipoest: (flojo, trabajador, puntual, inpuntual)  
END;

3 Estúdiense la definición y la declaración siguientes:

```
TYPE
fecha = RECORD
mes: 1..12;
día: 1..31;
año: 0..2001;
bisiesto: (sí, no)
END;
VAR tiempo: fecha;
```

Determíñese cuáles de las siguientes asignaciones son válidas:

- a) tiempo.mes := 12;
- b) tiempo.fecha := 3-15-1900;
- c) tiempo.fecha := mes;
- d) tiempo.año := 2002;
- e) tiempo.bisiesto := 1;
- f) tiempo.día := tiempo.mes;
- g) tiempo.tiempo := tiempo;

4 Examinense la definición y la declaración siguientes:

```
TYPE
info = RECORD
nombre: PACKED ARRAY [1..30] OF char;
edad: 1..99;
```

```
fechanac: PACKED ARRAY [1..10] OF char;  
END;
```

VAR

```
persona: info;
```

Determine cuáles de las siguientes proposiciones son válidas:

- a) persona.nombre := persona fechanac;
- b) persona.nombre [30] := persona.fechanac[10];
- c) persona.edad := persona.edad + 1;
- d) persona.fechanac := '12-12-1912';
- e) persona.fechanac[11] := ':';

5 Con la declaración de registro del problema 4, esribase un procedimiento para leer uno de esos registros.

6 Con la declaración de registro del problema 4, esribase un procedimiento para exhibir uno de esos registros con encabezados adecuados.

## SECCIÓN 11.2 REGISTROS JERÁRQUICOS Y ARREGLOS DE REGISTROS

En muchas aplicaciones los registros pueden ser tan sólo componentes de una estructura de datos mayor. Por ejemplo, en un sistema de expedientes de alumnos, es posible que el registro de cada estudiante sea un componente de un arreglo de registros de estudiantes. Además, puede ser que dentro de cada registro de estudiante exista un componente que sea también registro. En esta sección se mostrará que un componente de un registro puede ser un registro (registros anidados) y que el tipo de componente de un arreglo puede ser un registro (arreglo de registros).

### Registros jerárquicos

Los registros que tienen un identificador de campo cuyo tipo de datos sea también un registro se denominan *registros jerárquicos* o *registros anidados*. Por ejemplo, el registro de un estudiante puede contener un componente que incluya la experiencia universitaria previa del estudiante.

TYPE

```
univprevia = RECORD  
    nombre: PACKED ARRAY [1..75] OF char;  
    dirección: PACKED ARRAY [1..50] OF char;  
    crédprevios : 0..máxint;  
    prom: real  
END;
```

Dada la definición del registro *univprevia*, el registro completo del estudiante se definiría así:

## TYPE

```

alumno = RECORD
  nombre: PACKED ARRAY [1..50] OF char;
  año: (primero, segundo, tercero, cuarto);
  educprev: univprevia;
  prom: real;
  créditos: 0..máxint
END;

```

## VAR

```
persona: alumno;
```

En un registro, los identificadores de campo tienen alcance limitado. Esto permite el uso del mismo identificador en el programa principal o dentro de otros registros. Obsérvese que en el ejemplo se usan los identificadores de campo *nombre* y *prom* en ambos registros. Dentro de un registro dado, los identificadores de campo deben ser únicos, pero se pueden emplear en otros registros.

Otra forma de escribir la definición del registro jerárquico *alumno* es la siguiente:

## TYPE

```

alumno = RECORD
  nombre: PACKED ARRAY [1..50] OF char;
  año: (primero, segundo, tercero, cuarto);
  educprev: RECORD
    nombre: PACKED ARRAY [1..75] OF char;
    dirección: PACKED ARRAY [1..50] OF char;
    crédprevios : 0..máxint;
    prom: real
  END;
  prom: real;
  créditos: 0..máxint
END;

```

## VAR

```
persona; alumno;
```

Supóngase que se desea tener acceso al promedio previo del estudiante (*prom*) y almacenar el resultado en una variable real llamada *califprev*. La proposición de asignación

```
califprev := persona.educprev.prom;
```

cumple este cometido. Dado que el registro está anidado, es preciso emplear el punto dos veces. La siguiente proposición exhibe el número de créditos obtenidos previamente:

```
writeln ('Créditos previos:', persona.educprevia.crédprevios:1)
```

y esta proposición asigna el valor cero a los créditos previos:

Cuando se manejan registros jerárquicos, conviene verificar el tipo de datos del componente al que se tiene acceso. Recuérdese que al hacer asignaciones o evaluar expresiones, el Pascal verifica automáticamente los tipos.

## Arreglos de registros

Ahora se examinarán los arreglos. Recuérdese que el tipo de los componentes de un arreglo puede ser cualquier tipo de datos, incluso tipos de datos estructurados como son los arreglos y registros. Por ejemplo, es posible que un empleado tenga su registro individual. Sin embargo, puede ser que en la empresa trabajen cien empleados. Pascal permite definir un arreglo de 100 registros de esta manera (obsérvese que la dirección también es un registro):

**TYPE**

```
info = RECORD
    nombre: PACKED ARRAY [1..35] OF char;
    dirección: RFCORD
        calle: PACKED ARRAY [1..30] OF char;
        ciudad: PACKED ARRAY [1..30] OF char;
        estado: PACKED ARRAY [1..2] OF char;
        códpost: 0..99999
    END;
    númss: PACKED ARRAY [1..11] OF char;
    edad: 16..99;
    exenciones: 0..máxint
END;
lista = ARRAY [1..100] OF info;
```

**VAR**

```
    empleado : lista;
```

La figura 11-2 muestra el arreglo de registros con datos para el primer empleado (es decir, *empleado[1]*).

La variable *empleado* es un arreglo de 100 elementos cuyos componentes son registros del tipo *info*. La expresión

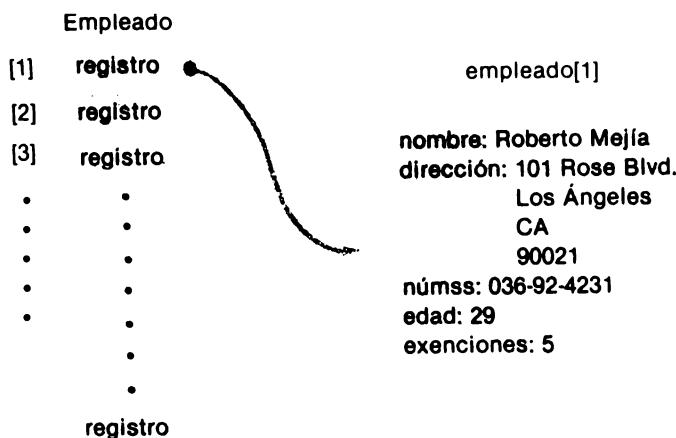
*empleado[1].edad*

permitirá tener acceso a la edad del primer empleado, y

*empleado[100].númss*

es la expresión que produce el número de seguro social del último empleado. Para exhibir los tres primeros dígitos de este número se podría usar el código

```
FOR índice := 1 TO 3 DO
    write (empleado[100].númss[índice])
```

**Figura 11-2** Arreglo de registros.

donde *índice* es una variable entera.

He aquí algunos ejemplos adicionales de proposiciones de asignación que incluyen el arreglo *empleado*.

```

empleado[1].nombre[1] := 'R';
empleado[5].exenciones := 0;
empleado[3].dirección.códpost := 68114;
empleado[100].dirección.estado := 'NL';
empleado[30].númss := '516-08-5049';

```

La siguiente tabla contiene expresiones y los tipos de datos correspondientes. Se recomienda al lector estudiar con cuidado esta tabla porque puede ayudarle a determinar el tipo de expresiones similares.

expresión	tipo de datos
empleado[1].nombre	PACKED ARRAY [1..35] OF char
empleado[1].nombre[1]	char
empleado[1].dirección	RECORD . . .
empleado[3].númss	PACKED ARRAY [1..11] OF char
empleado[3].dirección.códpost	0..99999
empleado[3].dirección.estado	PACRED ARRAY [1..2] OF char
empleado[4].dirección.estado[1]	char
empleado[5].edad	16..99

### Proposición WITH

En el ejemplo anterior, supóngase que se desea hacer las siguientes modificaciones en el registro del primer empleado:

```

empleado[1].edad := empleado[1].edad + 1;
empleado[1].exenciones := empleado[1].exenciones -1;

```

Se ha repetido el identificador del registro en todas las proposiciones de asignación, pero repetirlo tantas veces puede consumir mucho tiempo y propiciar errores. Pascal permite abreviar estas asignaciones y otras proposiciones que hacen referencia a componentes similares mediante una proposición WITH (con):

```
WITH empleado[1] DO  
BEGIN  
    edad := edad + 1;  
    exenciones := exenciones - 1;  
    dirección.calle := 'Ave. Maple Núm. 35';  
    dirección.estado[1] := 'V';  
END
```

La palabra reservada WITH va seguida del nombre de un identificador de registro (o una lista de identificadores de registro separados por comas) seguido a su vez de una proposición o grupo de proposiciones enmarcadas por las palabras reservadas BEGIN y END. La forma general de la proposición WITH es

#### WITH lista de identificadores de registro DO proposición

En la “proposición” que aparece en la proposición WITH, las referencias a un componente de la variable de registro nombrada se pueden abreviar escribiendo solamente la parte del indicador de registro que aparece después de la variable de registro. Por ejemplo, considérense las siguientes variables:

```
VAR  
    reg: RECORD  
        c1 : integer;  
        c2 : real;  
        c3 : char  
    END;  
    c3 : char;      (* no es igual a reg.c3 *)  
    cosa : integer;
```

Así, las proposiciones

```
reg.c1 := cosa;  
reg.c2 := 1.0/reg.c2  
reg.c3 := 'X';
```

se pueden sustituir por

```
WITH reg DO  
BEGIN  
    c1 := cosa;  
    c2 := 1.0/c2;  
    c3 := 'X'  
END
```

Pero nótese que las proposiciones

```
reg.c1 := cosa;
reg.c2 := 1.0/reg.c2;
c3 := 'X';
```

*no se pueden sustituir por*

```
WITH reg DO
BEGIN
    c1 := cosa;
    c2 := 1.0/c2;
    c3 := 'X'
END
```

porque la proposición `c3 := 'X'` se interpretaría de tal modo que equivaldría a `reg.c3 := 'X'`. Obsérvese que es posible hacer referencia a la variable `cosa` sin problema en las proposiciones encerradas en el `WITH`, ya que no aparece como identificador de campo en la variable de registro especificada. Si así fuera, no sería posible utilizar la forma de referencia abreviada que permite la proposición `WITH`, y sería preciso escribir el indicador de campo completo. El diagrama de sintaxis de la proposición `WITH` se muestra en la figura 11-3.

Si se desea hacer referencia varias veces a denominadores de campo de registros jerárquicos o anidados, se pueden emplear proposiciones `WITH` anidadas. Considerese el siguiente segmento de programa:

```
WITH empleado[1] DO
    WITH dirección DO
        códpost := 68182;
```

Esto equivale a la proposición

```
empleado[1].dirección.códpost := 68182
```

que es lo que se escribiría normalmente, a menos que se tuviera un grupo de proposiciones que hicieran referencia a `empleado[1].dirección`. En ese caso sería preferible usar las proposiciones `WITH` con una pareja `BEGIN-END` para enmarcar al grupo de proposiciones con las referencias. Se pueden escribir las proposiciones `WITH` anidadas que se mostraron antes así:

```
WITH empleado[1].dirección DO
    códpost := 68182
```

**Figura 11-3** Diagrama de sintaxis de la proposición `WITH`.



```
WITH empleado[1].dirección DO
    códpost := 68182
```

Adviértase que es importante el orden en que se escriben *empleado[1]* y *dirección* en la proposición WITH. La siguiente proposición es errónea:

```
WITH dirección, empleado[1] DO
    códpost := 68182
```

En general, la notación

```
WITH reg1, reg2, reg3 DO
    proposición;
```

significa exactamente lo mismo que

```
WITH reg1 DO
    WITH reg2 DO
        WITH reg3 DO
            proposición;
```

Supóngase que se tienen dos identificadores de campo con el mismo nombre pero de diferentes registros y se hace referencia a ellos en la misma proposición WITH. ¿Cuál de las proposiciones WITH anidadas proporcionará la identidad de la variable de registro? Como se podría esperar, la variable de registro más interior tendrá prioridad sobre las demás. Esto se debe, básicamente, a la misma regla que se usa cuando una variable local (en un procedimiento o función) tiene el mismo nombre que una variable global.

En la sección de técnicas de prueba y depuración se examinarán algunos problemas que pueden presentarse al usar proposiciones WITH.

## EJERCICIOS DE LA SECCIÓN 11.2

1 Estúdiense las siguientes declaraciones:

```
TYPE
    tipofecha = RECORD
        mes: PACKED ARRAY [1..3] OF char;
        día: 1..31;
        año: 1900..2000;
        hora: 0..2399
    END;
    tiponombre = RECORD
        apellido: PACKED ARRAY [1..20] OF char;
        nombila: PACKED ARRAY [1..10] OF char
    END;
```

```

paciente = RECORD
    nombre: tiponombre;
    sexo: (masc, fem);
    edocivil: (soltero, casado);
    doctor: tiponombre;
    cita: tipofecha
END;

VAR
    internado,
    convaleciente: ARRAY [1..20] OF paciente;

```

Determine cuáles de las siguientes asignaciones son válidas:

- a) internado[5].nombre := 'Juan Pérez';
- b) internado[1].sexo := fem;
- c) internado[6].cita.mes := 'Jun';
- d) convaleciente[2].paciente := internado[1].paciente;
- e) convaleciente[2].edad := 0;
- f) convaleciente := Internado;
- g) internado[0].seguro := true;
- h) internado[1].doctor.nompila := 'Fernando'

2 Léanse las siguientes declaraciones:

```

TYPE
    palabra: ARRAY [1..25] OF char;
    estructura = RECORD
        valor: integer;
        legal: Boolean;
    END;
    lista = RECORD
        bloque: ARRAY [1..2000] OF palabra;
        símbolo: ARRAY [1..20] OF palabra
    END;
    códreg = RECORD
        códpal: palabra;
        código: (A, B, C, D, E, L, M);
        diagrama: lista
    END;

VAR
    bloquecód: ARRAY [1..200] OF códreg;
    tabla: estructura;
    complejo: ARRAY [1..80] OF estructura;
    uncódreg: códreg;
    unalista: lista;
    i, cuenta: integer;
    unapal: palabra;

```

- Determine cuáles de las siguientes proposiciones son válidas:
- bloquecód[6].diagrama := unalista;
  - uncódreg.códpal[1] := unalista.bloque[1];
  - bloquecód[100].diagrama.bloque[1,2] := unapal [2];
  - IF bloqucód[20].código = L  
THEN bloquecód[20].diagrama.bloque[1,1] := 'L';
  - uncódreg.valor := i;
  - IF legal  
THEN cuenta := i;
  - IF códreg.código = M  
THEN cuenta := cuenta + 1;
  - bloquecód[1].diagrama.bloque[2] := unapal;
  - complejo[100].valor := i;
  - bloquecód[20].bloque[1] := unapal;

3 Examíñese la siguiente declaración:

```
TYPE
    persona = RECORD
        nombre: PACKED ARRAY [1..20] OF char;
        edad: 1..99;
        estado: (casado, soltero, divorciado);
        salario: real;
        exenciones: 0..máxint
    END;
VAR
    juanita: persona;
    grupo: ARRAY [1..100] OF persona;
```

Determine cuáles de las siguientes proposiciones son válidas:

- grupo[1] := juanita;
- grupo[1].nombre := 'juanita';
- read (grupo[1].estado);
- WITH grupo DO  
writeln (nombre);
- WITH grupo[100] DO  
BEGIN  
writeln (nombre);  
read (edad)
- END;  
WITH juanita DO  
BEGIN  
nombre := grupo[50].nombre;  
salario := grupo [1].salario

END;

4 Con las declaraciones del problema 3, determine si el siguiente segmento de código es válido (se supone que *índice* es una variable entera):

```

FOR índice := 1 TO 100 DO
    writeln (grupo[101-índice].nombre);
FOR índice := 1 TO 100 DO
    CASE estado OF
        casado: writeln (grupo[índice].salario);
        soltero: writeln (grupo[índice].nombre);
        divorciado: writeln (grupo[índice].exenciones)
    END;

```

- 5 Dadas las declaraciones del problema 3 y una variable entera llamada *índice*, supóngase que el nombre contenido en *grupo[1].nombre* es

' T. J. Booker

y el nombre contenido en *grupo[2].nombre*, *grupo[3].nombre*, ..., *grupo[100].nombre* es

'Humpty Dumpty

Determínese qué es lo que exhibe el siguiente segmento en Pascal:

```

índice := 1;
WITH grupo[índice] DO
    WHILE índice < = 100 DO
        BEGIN
            writeln (nombre);
            índice := índice + 1
        END;

```

- 6 Repítase el problema 5 con este otro segmento en Pascal:

```

índice := 1;
WHILE índice < = 100 DO
    WITH grupo[índice] DO
        BEGIN
            writeln (nombre);
            índice := índice + 1
        END;

```

### SECCIÓN 11.3 RESOLUCIÓN DE PROBLEMAS MEDIANTE REGISTROS

En esta sección se examinará un problema común en el que se emplean registros. El problema se refiere a la clasificación de un arreglo de registros.

#### Clasificación de un arreglo de registros

¿Se ha preguntado alguna vez el lector por qué muchas veces se le pide que escriba primero su apellido paterno al llenar una forma? Como ya se aprendió en un

capítulo anterior, cuando los datos están clasificados es posible tener acceso a ellos mucho más rápidamente (p. ej., mediante el uso de una búsqueda binaria) que cuando los datos están desordenados. El directorio telefónico que usa el lector está clasificado en orden alfabético (técnicamente, en orden *lexicográfico*). Obsérvese que en el directorio telefónico casi siempre aparece primero el apellido paterno, seguido del materno y los nombres de pila. Cuando se proporciona primero el apellido paterno se contribuye a facilitar la clasificación de una lista de nombres.

## PROBLEMA 11.2

*Escribase un programa que clasifique alfabéticamente por apellido un arreglo de registros que contienen datos sobre cien estudiantes de la carrera de computación. Para simplificar el problema, supóngase que todos los apellidos son distintos. Si no fuera así, sería preciso clasificar también según los nombres de pila. Supóngase que los registros tienen la siguiente estructura:*

Apellido paterno: 30 caracteres máximo  
Nombre propio: 30 caracteres máximo  
Inicial del apellido materno: un solo carácter  
Año: primero, segundo, tercero, cuarto  
Promedio: número real  
Créditos en computación: entero

La definición de registro y las declaraciones de variables serán las que se muestran a continuación. El nombre del estudiante se definió también como registro.

### TYPE

```
normalum = RECORD
    paterno: PACKED ARRAY [1..30] OF char;
    nompila: PACKED ARRAY [1..30] OF char;
    materno: char
END;
alumno = RECORD
    nombre: normalum;
    año: (primero, segundo, tercero, cuarto);
    prom: real;
    crédcomp: 0..máxint
END;
lista = ARRAY [1..100] OF alumno;
```

### VAR

```
alumcomp: lista;
```

Para clasificar ahora el arreglo de registros, la llave debe ser el componente que contiene el apellido paterno de un estudiante determinado (cuyo índice es *i*):

**alumcomp[i].nombre.paterno**

Este componente es un arreglo empacado de 30 caracteres. El componente *paterno* se denomina *llave del registro*. Anteriormente se escribió un procedimiento para clasificar enteros en orden ascendente. En este caso será preciso clasificar cadenas de caracteres. Puesto que se declaró *paterno* como arreglo empacado, es posible emplear los operadores relacionales para comparar apellidos igual que se hizo con los enteros. Para presentar al lector otras técnicas de clasificación se utilizará un procedimiento diferente llama *clasificación de burbuja*.

### Clasificación de burbuja

Muchos métodos se basan en intercambiar parejas de elementos que están en desorden hasta que todas están en orden. La clasificación de burbuja es una técnica muy conocida (pero no demasiado eficiente) que emplea este método. La idea del algoritmo es la siguiente: examinar parejas adyacentes de nombres de izquierda a derecha, repetidamente, intercambiando todas las que no estén en orden. Considérese el siguiente ejemplo de palabras de tres letras que se deben clasificar:

DAR	CON	VER	TEN	IVA	IBA
					

Partiendo de la izquierda y avanzando hacia la derecha, se comparan los elementos de las parejas hasta que se encuentra una pareja en desorden. Inmediatamente se ve que *DAR* y *CON* no están en orden, por lo que se intercambian:

CON	DAR	VER	TEN	IVA	IBA
					

Al continuar se ve que *DAR* y *VER* están en el orden correcto, pero *VER* y *TEN* no, por lo que se intercambian:

CON	DAR	TEN	VER	IVA	IBA
					

En seguida se continúa, intercambiando *VER* e *IVA* y después *VER* e *IBA*. Después de completar este primer recorrido por el arreglo se tiene el orden

CON	DAR	TEN	IVA	IBA	VER

Obsérvese que el valor más alto ha “suñido como burbuja” hasta el extremo derecho del arreglo. Si se repite este proceso (es decir, se hace otro recorrido), el siguiente valor más alto subirá hasta su posición correcta:

CON	DAR	IVA	IBA	TEN	VER

El siguiente elemento más grande, *TEN*, ha subido a su posición correcta, justo antes de *VER*. Puesto que, después del primer recorrido, el elemento más grande se pasó a su posición correcta, no fue necesario examinar la última pareja de val-

res: ya se sabía que estaba en orden. Después del siguiente recorrido, no es preciso examinar las últimas dos parejas, ya que deben estar en orden. De manera similar, se sabe que después del recorrido número  $I$  por el arreglo no será necesario examinar las parejas que contienen los últimos  $I$  elementos, ya que se garantiza su orden.

Téngase en cuenta que si son  $N$  los elementos del arreglo que se van a ordenar, basta con hacer  $N-1$  recorridos de los datos. Recuérdese que después de cada recorrido se pasa otro dato a su posición correcta. Por tanto, después de  $N-1$  recorridos,  $N-1$  datos estarán en su posición correcta dentro del arreglo. ¿Qué sucede entonces con el último dato? Que forzosamente debe estar en el lugar correcto. El arreglo del ejemplo queda completamente clasificado después de tres recorridos:

CON      DAR      IBA      IVA      TEN      VER

En seguida se muestra el algoritmo de la clasificación de burbuja, donde *nombre* es un arreglo de  $N$  elementos. Obsérvese que se necesita un ciclo anidado. El ciclo interior (FOR) compara parejas de valores. El ciclo exterior (WHILE) repite este proceso para los valores restantes que no están todavía en sus posiciones correctas hasta que el arreglo queda clasificado.

#### *Algoritmo de clasificación de burbuja*

Sea *control* igual  $N$  (tamaño del arreglo).

Mientras *control* no es igual a cero hacer:

    Sea *temp* igual a cero;

    Para *indice* = 1 hasta *control*-1 hacer:

        Si *nombre*[*indice*] > *nombre*[*indice* + 1]

            Intercambiar *nombre*[*indice*] y *nombre*[*indice* + 1];

            Sea *temp* igual a *indice*.

    Sea *control* igual a *temp*.

Se ha mejorado el algoritmo arriba descrito. Se asigna el valor cero a la variable *temp* antes de cada recorrido y cuando se realiza un intercambio se asigna a *temp* el valor del índice del primer elemento de la pareja que se intercambió. En el siguiente recorrido se examinan las parejas únicamente hasta la posición del último intercambio en el recorrido anterior, *ya que todas las parejas mas allá de ese punto deben estar en orden*. Si no lo estuvieran, *temp* indicaría el punto en el que se hizo el intercambio. Se recomienda al lector estudiar este algoritmo con mucho cuidado y verificar que es correcto mediante su aplicación al ejemplo de palabras de tres letras que se usó antes.

### Resolución del problema

ciso intercambiar los registros en los que aparecen los nombres. Si no se hiciera así, el resto de la información referente a los estudiantes cuyos registros se intercambiaron quedaría en registros erróneos. El siguiente programa realiza las tareas necesarias e incluye las modificaciones apropiadas al algoritmo de clasificación de burbuja. Obsérvese que es preciso declarar un registro temporal para intercambiar dos registros del arreglo. El programa incluye la invocación de dos procedimientos, *leer registros* y *ponregistros*, que no se detallan. Son tema del ejercicio 4 al final del capítulo.

```

PROGRAM clasifreg (input, output);
(* Leer, clasificar por apellido paterno y exhibir los registros *)
(* de hasta 100 estudiantes de la carrera de computación. *)
CONST
  tammáx = 100;
TYPE
  nomalum = RECORD
    paterno: PACKED ARRAY [1..30] OF char;
    nompila: PACKED ARRAY [1..30] OF char;
    materno: char
  END;
  alumno = RECORD
    nombre: nomalum;
    año: (primero, segundo, tercero, cuarto);
    prom: real;
    crédcomp: 0..máxint
  END;
  lista = ARRAY [1..tammáx] OF alumno;
  tamtipo = 0..tammáx;
VAR
  alumcomp: lista;
  tamaño: tamtipo; (* número de registros *)
PROCEDURE leerregistros;
(* Leer los registros y determinar tamaño *)
(* véase ejercicio 4 al final del capítulo *)
PROCEDURE ponregistros;
(* véase ejercicio 4 al final del capítulo *)
PROCEDURE clasifburbuja (VAR arreglos: lista; tamaño: tamtipo);
(* Clasificar arreglo de registros por apellido paterno *)
(* usando la clasificación de burbuja *)
VAR
  control, índice, temp: 1..tammáx;
  regtemp: alumno; (* Registro temporal *)
BEGIN
  control := tamaño;
  WHILE control <> 0 DO (* mientras no esté clasificado el arreglo *)
    BEGIN
      temp:=0;
      FOR índice:= 1 TO control-1 DO (* con todas las parejas * )

```

arreglos[índice + 1].nombre. paterno.

THEN BEGIN

(\* no, intercambiar registros \*)

regtemp := arreglos[índice];

arreglos[índice] := arreglos[índice + 1];

arreglos[índice + 1] := regtemp;

(\* guardar índice de última pareja intercambiada \*)

temp := índice

END,

ontrol := temp (\* Ahora examinar sólo esa parte del arreglo, \*)

END (\* desde uno hasta la última pareja intercambiada \*)

END; (\* clasifburbuja \*)

(\* Programa principal \*)

BEGIN

leer registros; (\* leer registros de los estudiantes \*)

clasifburbuja (alumcomp, tamaño); (\* clasificar registros \*)

ponregistros (\* exhibir registros ya clasificados \*)

END;

## SECCIÓN 11.4 REGISTROS VARIANTES [OPCIONAL]

En esta sección se presenta una breve introducción a los registros variantes. Pascal permite que los registros incluyan componentes que pueden variar de un registro a otro. Los registros que tienen dos partes separadas, una parte *fija* y otra parte *variante*, se llaman *registros variantes*. La parte fija del registro consta de aquellos componentes (o *campos*) comunes a todas las variables de registro del tipo dado. La parte variante declara los campos que pueden existir según el valor de un determinado campo (llamado *campo etiqueta*). Por ejemplo, supóngase que se tiene un sistema de registros de estudiantes que contiene el nombre del alumno, dirección, promedio, total de créditos y situación. Según sea la situación

```

    prom: real;
    créditos: 0..máxint;
CASE actual: situación OF
    primero: (promprep: real);
    transferido: (crédtrans: 0..máxint);
    graduado: (añograd: 1900..2000);
    otro: ( )
END

```

La parte fija del registro contiene nombre, dirección, promedio y créditos del estudiante; siempre se coloca antes de la parte variante del registro. El campo *actual* se denomina *campo etiqueta*; en este ejemplo el *tipo de la etiqueta* es *situación*. El campo etiqueta sirve para almacenar un valor que indica cuáles de los campos variantes estarán presentes en el registro variante. Obsérvese que la parte variante del registro es similar a una proposición CASE, pero no idéntica. La palabra reservada CASE va seguida de la declaración del componente etiqueta (*campo etiqueta: tipo etiqueta*), la palabra reservada OF y los campos variantes. Cada uno de los campos variantes contiene una lista de una o más constantes, un signo de dos puntos y la definición del campo variante encerrada entre paréntesis. Puesto que la parte variante del registro debe aparecer al final del registro, sólo se requiere un END para finalizar su definición. Los identificadores que se usan en los campos variantes deben ser diferentes de los utilizados en la parte fija del registro.

Los registros variantes sólo pueden tener una parte variante. Sin embargo, una parte variante puede incluir su propia parte variante (es decir, se permiten las variantes anidadas). Examíñese el siguiente ejemplo:

```

TYPE
    cadena = PACKED ARRAY [1..25] OF char;
    tipocomp = (analógica, digital);
    configuración = (independiente, híbrida);
    reganálog = RECORD
        CASE config: configuración OF
            independiente: ();
            híbrida: (partedigital: cadena)
        END;
    regdigital = RECORD
        CASE config: configuración OF
            independiente: ();
            híbrida: (parteanalógica: cadena)
        END;
    computadora = RECORD
        modelo: cadena;
        fabricante: cadena;
        CASE tipoc: tipocomp OF
            analógica: (reganálog: reganálog);
            digital: (regd: regdigital)
        END
    END

```

En este caso se espera que cada computadora tenga un nombre de modelo, un nombre de fabricante y un tipo (ya sea *análogica* o *digital*). Las computadoras híbridas son sistemas de computadoras analógicas y digitales conectadas entre sí, por lo que si una computadora es parte de sistema híbrido, el registro incluye el nombre de la otra computadora del sistema.

La variante específica de un registro a la que se haga referencia se determina durante la ejecución del programa. Por ejemplo, supóngase que se hace la declaración

**VAR**

**persona: estudiante**

Se puede hacer referencia a los componentes fijos de la variable de registro *persona* de la manera usual: *persona.nombre*, *persona.prom* y así sucesivamente. Para tener acceso a un campo de la parte variante, el campo variante (*actual*) debe contener un valor que seleccione el campo variante deseado. Por ejemplo, la proposición de asignación

**persona.actual := primero**

especifica que la variante del registro persona que se usará es la que contiene el promedio obtenido en preparatoria (*promprep*). Ahora ya se puede tener acceso a *persona.promprep*. Por ejemplo, la proposición

**persona.promprep := 3.10**

es válida. Esta asignación no se hubiera permitido si la variable del campo etiqueta no tuviera el valor apropiado. Por ejemplo, la asignación

**persona.añograd := 1985**

no es válida a menos que el valor actual del campo etiqueta sea *graduado*. Para asignar el valor *graduado* al campo etiqueta se puede usar la proposición de asignación

**persona.actual := graduado**

Supóngase que se declaran dos variables del tipo de registro *estudiante*:

**VAR**

**persona1, persona2: estudiante;**

Por las asignaciones que se hacen en seguida puede verse que las variables *persona1* y *persona2* son de tipo de registro idéntico, pero tienen estructuras diferentes. La parte variante está vacía en *persona2*, cosa que no sucede en *persona1*. Estas asignaciones son válidas para *persona1*:

**persona1.nombre := 'Juan Bonaparte'**  
**persona1.prom := 3.7;**

```
persona1.actual := transferido;
persona1.crédtrans := 5
```

y estas otras son válidas para *persona2*:

```
persona2.nombre := 'María T. Cordero
persona2.prom := 3.9;
persona2.actual := otro
```

## SECCIÓN 11.5 TÉCNICAS DE PRUEBA Y DEPURACIÓN

En esta sección se examinarán algunos errores comunes que se deben al mal uso de los registros. Considérese la siguiente definición de registro y declaración de variable:

**TYPE**

```
mal = RECORD
    nombre: PACKED ARRAY [1..30] OF char;
    sexo: (masc, fem);
    notas: ARRAY [1..5] OF integer
END;
lista = ARRAY [1..50] OF mal;
```

**VAR**

```
estudiante: lista;
```

Un error frecuente en la definición de registro es la omisión de la palabra reservada END para terminar la definición del registro. ¿Qué hará el compilador en esta situación? Seguiría procesando el programa fuente en busca de identificadores de campo seguidos del identificador de tipo (recuérdese el diagrama de sintaxis del tipo de registro). Esto conduce normalmente a un número más o menos grande de errores, el cual depende de la sintaxis y la capacidad del compilador para recuperarse de los errores; la mayor parte de los compiladores tratarán de continuar con la compilación si es posible.

Otro error muy común al usar registros es emplear el identificador de tipo en vez de la variable de registro. Con el mismo ejemplo, he aquí algunos casos:

error de registro	corregido	razón
estudiante[1].mal	estudiante[1].nombre	El identificador de registro es <i>mal</i> .
estudiante[1].lista	estudiante[1].nombre	El tipo del arreglo es <i>lista</i> .
estudiante.nombre	estudiante[1].nombre	Falta el índice.

Se presentan errores cuando el tipo del identificador de campo no concuerda con el tipo de una variable o constante asignada. Por ejemplo, la siguiente proposición de asignación no es válida (supóngase que *nuevanota* es una variable entera):

```
estudiante[1].notas := nuevanota;
```

```
estudiante[1].notas[1] := nuevanota;
```

Las variables de registro pueden servir como parámetros verdaderos que correspondan a parámetros variables de funciones y procedimientos, pero no sucede lo mismo con los campos etiqueta de los registros variantes. Si se piensa en la variable de registro *persona* de la sección anterior, *persona.nombre* se podría pasar como parámetro verdadero (a un parámetro variable formal de tipo arreglo empacado), no así *persona.promprep*, ya que es un componente de la parte variante del registro. Una forma de resolver este problema es copiar el componente variable a una segunda variable (que no sea parte de un arreglo empacado o una estructura) y usar esta segunda variable como parámetro verdadero. El valor de la segunda variable después de ejecutarse el procedimiento o función se deberá reasignar al componente variante.

En seguida se considerarán los errores que se pueden presentar cuando se usa la proposición WITH. Supóngase que se agrega la siguiente declaración de variable al ejemplo:

VAR

```
    sexo: char;
```

En la definición de registro aparece también *sexo* como variable de tipo enumerado. Por tanto, la proposición

```
WITH estudiante[1] DO  
    sexo := 'M'
```

no es válida, ya que el identificador *sexo* dentro de la proposición WITH la hace equivalente a la proposición

```
estudiante[1].sexo := 'M'
```

y 'M' no es una constante del tipo apropiado (enumerado). Las proposiciones que hagan referencia a la variable de carácter *sexo* no deben estar dentro del alcance de una proposición WITH que especifique un elemento del arreglo de registros *estudiante*. Las proposiciones

```
sexo := 'M';  
WITH estudiante[1] DO  
    sexo := masc
```

sí están permitidas.

En los ejemplos anteriores fue necesario emplear un subíndice después del nombre del arreglo (*estudiante*). Esto se debe a que la proposición WITH requiere la especificación de una variable de registro y no permitirá el uso de un arreglo de registros (véase el diagrama de sintaxis de la figura 11-3). Así, la siguiente proposición WITH es incorrecta:

**WITH estudiante DO****[1].sexo := masc;**

Un último problema que puede surgir al usar la proposición WITH se relaciona con el empleo de una variable para seleccionar un componente de un arreglo de registros. Considérese el siguiente segmento de código;

```
índice := 1;
```

```
WITH estudiante[índice] DO
```

```
FOR índice := 1 TO 50 DO
```

```
writeln ('Nombre del estudiante', índice:1, ':' nombre);
```

En este caso el nombre que está en *estudiante[1]* se exhibirá 50 veces. Los primeros renglones de la salida tendrán este aspecto:

**Nombre del estudiante 1: Juan Pérez**

**Nombre del estudiante 2: Juan Pérez**

**Nombre del estudiante 3: Juan Pérez**

El valor de *índice* cambiará, como puede verse, pero el componente *nombre* no. Esto se debe a que el registro específico al que se tiene acceso en la proposición WITH queda determinado por la variable de registro especificada *en el momento en que se ejecuta la proposición WITH*. Por tanto, el valor que tenía *índice* antes de la proposición WITH es el que fija el nombre que se va a exhibir. El código correcto para exhibir los nombres de todos los estudiantes es

```
FOR índice := 1 TO 50 DO
```

```
WITH estudiante[índice] DO
```

```
writeln ('Nombre del estudiante', índice:1, ':' nombre);
```

He aquí una lista de recordatorios importantes de Pascal que pueden ser útiles para la prueba y depuración de programas:

## RECORDATORIOS DE PASCAL

- La lista de campos en la definición de un registro termina con la palabra reservada END.
- Los identificadores de campo en un registro deben ser únicos. Otros registros pueden volver a usar los mismos identificadores.
- Para tener acceso a los componentes de un registro se usan denominadores de campo. La variable de registro y el identificador de campo se separan mediante un punto:

**estudiante[1]. nombre**

- Las reglas de lectura y exhibición de registros se aplican a cada uno de los denominadores de campo con base en el tipo del denominador de campo.
- Se pueden asignar registros completos a otros registros completos de tipo idéntico.
- Los registros pueden estar anidados (registros jerárquicos).

- Los denominadores de campo tienen tipos. Éstos deben satisfacer los mismos requisitos que los tipos de las variables y constantes cuando se emplean en expresiones o proposiciones de asignación.
- La proposición WITH debe especificar una variable de registro.
- La parte variante de un registro debe aparecer después de la parte fija.
- Para declarar la parte variante de un registro se usa una cláusula CASE. La palabra clave CASE va seguida inmediatamente de una declaración de campo etiqueta. Las variantes se incluyen en forma de lista de campos entre paréntesis precedidos de una lista de constantes y un signo de dos puntos.
- El valor del campo etiqueta determina cuál de las variantes está presente.

## SECCIÓN 11.6 REPASO DEL CAPÍTULO

En este capítulo se analizó el tipo de datos estructurado conocido como registro. La propiedad característica de los registros es que los componentes pueden ser de diferentes tipos de datos. Los registros se definen mediante una lista de identificadores de campo, seguidos de los identificadores de tipo correspondientes. Se demostró la manera de elaborar estructuras de datos complejas mediante la combinación de registros y arreglos para formar las siguientes estructuras:

- Registro de arreglos (los componentes de un registro pueden ser arreglos)
- Arreglo de registros (los componentes de un arreglo pueden ser registros)
- Registro de registros (los componentes de un registro pueden ser registros)

El uso de la proposición WITH permite simplificar el acceso a los componentes de estructuras complejas, como son los arreglos de registros anidados.

En este capítulo se presentó también el problema de clasificar un arreglo de registros. Se demostró la forma de usar el algoritmo de clasificación de burbuja para ordenar un arreglo de registros de estudiantes según el apellido paterno de los alumnos. En una sección opcional se analizó la declaración y el acceso a los componentes de registros variantes. En la sección de técnicas de prueba y depuración se incluyen varios ejemplos de errores comunes que se presentan cuando se usan registros.

En seguida se presenta un resumen de las características de Pascal que se estudiaron en este capítulo con el fin de contar con una referencia rápida en el futuro.

## REFERENCIAS DE PASCAL

- 1 Registro: tipo de datos estructurado cuyos componentes pueden tener diferente tipo.

Forma general:

**TYPE**

```
identificador = RECORD
    identificador de campo: tipo;
    identificador de campo: tipo;
    ...
    identificador de campo: tipo
END;
```

Ejemplo:

**TYPE**

```
    empleado = RECORD
        nombre: PACKED ARRAY [1..30] OF char;
        salario: 0..máxint;
        sexo: (masc, fem)
    END;
```

- 2 Denominador de campo: sirve para tener acceso a un componente gistro.

Ejemplo:

```
empleado.nombre[1]
```

- 3 Registros jerárquicos: tipos de registro anidados.

Ejemplo (se usa el tipo de registro *empleado* arriba definido):

**TYPE**

```
    depto = RECORD
        nombre: PACKED ARRAY [1..20] OF char;
        jefe: empleado;
        númdesp: integer;
    END;
```

- 4 Arreglo de registros: arreglo cuyos componentes son de tipo de registro.

Ejemplo:

**TYPE**

```
    lista : ARRAY [1..10] OF depto;
    VAR
        sección: lista;
```

- 5 Proposición WITH: las referencias a un denominador de campo en una proposición WITH se aplican a la variable de registro que sigue la palabra reservada WITH.

Forma general:

```
WITH identificador de registro DO
    proposición;
```

Ejemplo:

```
WITH sección[1] DO
    númdesp := 1
```

- 6 Registro variante: contiene una parte fija y una parte variante.

Forma general:

```

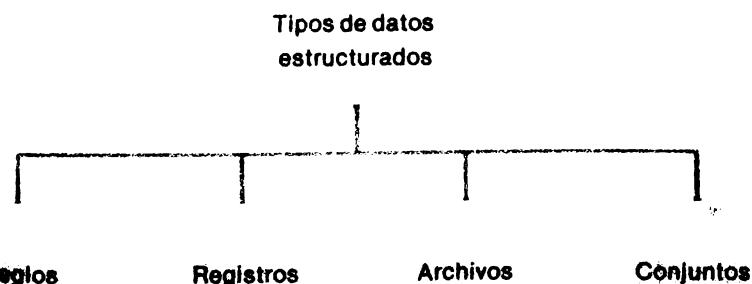
identificador = RECORD
  (* parte fija *)
  identificador de campo: tipo;
  identificador de campo: tipo;
  ...
  identificador de campo: tipo;
  (* parte variante *)
  CASE etiqueta: tipo etiqueta OF
    constante: (lista de campos);
    constante: (lista de campos);
    ...
    constante: (lista de campos);
END;

```

## Avance del capítulo 12

Pascal cuenta con cuatro tipos de datos estructurados: arreglos, registros, archivos y conjuntos (véase la Fig. 11-4). En el siguiente capítulo se analizarán las estructuras de archivo. Los archivos se parecen a los arreglos en tanto que todos los componentes son del mismo tipo. Los archivos, empero, se almacenan normalmente fuera de la memoria primaria de la computadora en dispositivos como discos y cintas magnéticas. El uso de archivos es ventajoso porque éstos no tienen un tamaño máximo previamente determinado, y su contenido tiene una duración diferente del periodo durante el cual se ejecuta el programa. Así, es posible almacenar grandes cantidades de datos sin la limitación del tamaño del almacenamiento primario. Las desventajas principales del uso de archivos para almacenar datos son las velocidades relativamente bajas de los dispositivos en que se almacenan y la restricción de Pascal estándar de que los componentes deben leerse en orden secuencial estricto.

**Figura 11-4** Tipos de datos estructurados.



**Palabras clave del capítulo 11**

arreglo de registros	orden lexicográfico
campo	parte fija
campo etiqueta	parte variante
clasificación de burbuja	proposición WITH
denominador de campo	registro
identificador de campo	registro variante
llave de registro	registros jerárquicos

**EJERCICIOS DEL CAPÍTULO 11****★ EJERCICIOS ESENCIALES**

- 1 Describábase mediante una definición TYPE adecuada el registro que sería apropiado para cada uno de los siguientes tipos de datos:
  - a) Los libros de una biblioteca personal.
  - b) Las partes de un objeto complejo. No debe ovidarse la necesidad de describir partes compuestas por otras partes.
  - c) La historia académica de un estudiante durante varios semestres de educación universitaria.
  - d) La definición de diccionario de una palabra cualquiera.
- 2 Supóngase que se tienen dos tipos de registro,  $T_1$  y  $T_2$ , que se definen en seguida. Es posible asignar variables completas del tipo  $T_1$  a otras variables de tipo  $T_1$ , y lo mismo puede decirse de las de tipo  $T_2$ . No es posible asignar una variable completa de tipo  $T_1$  a una de tipo  $T_2$ , aunque todos los identificadores de  $T_1$  aparezcan también como identificadores de campo en  $T_2$ . Escríbáse el código apropiado en Pascal para realizar esta tarea. Averíguese si el sistema de cómputo usado cuenta con lenguajes que permitan especificar una operación de asignación de este tipo.

**TYPE**

```

T1 = RECORD
  c1: integer;
  c2: real
END;
T2 = RECORD
  c1: integer;
  c2: real
  c3: char
END;
```

- 3 Los *números complejos* tienen dos componentes, una parte *real* y una parte *imaginaria*, cada una de las cuales se representa mediante un número real. Este tipo de registro es útil para representar números complejos:

```

complejo = RECORD
  re, im: real
END;

```

Escribanse procedimientos en Pascal con tres parámetros del tipo *complejo* (dos de valor y uno variable) que lleven a cabo la adición y la multiplicación de números complejos. Si *re1* e *im1* representan los componentes del segundo número complejo, entonces

```

re3 = re1 + re2
im3 = im1 + im2

```

especificará el resultado de la adición y

```

re3 = re1 * re2 - im1 * im2
im3 = im1 * re2 + im2 * re1

```

especificará el resultado de la multiplicación.

## ★ ★ EJERCICIOS IMPORTANTES

- 4 Escribanse los procedimientos *leerregistros* y *ponregistros* mencionados en el programa de la sección 11.3 Recuérdese que la estructura de los registros que se van a procesar se describe mediante estas definiciones:

TYPE

```

nomalum = RECORD
  paterno: PACKED ARRAY [1..30] OF char;
  nombila: PACKED ARRAY [1..30] OF char;
  materno: char
END;
alumno = RECORD
  nombre: nomalum;
  año: (primero, segundo, tercero, cuarto);
  prom: real;
  crédcamp: 0..máxint
END;
lista = ARRAY [1..100] OF alumno;

```

VAR

```

  alumcomp: lista;

```

Cada uno de los registros se representa externamente como una secuencia de campos de longitud variable separados entre sí mediante por lo menos un espacio en blanco. En cuanto al componente año del registro, se emplea un código entero, a saber:

- |   |         |
|---|---------|
| 1 | primero |
| 2 | segundo |
| 3 | tercero |
| 4 | cuarto  |

Por tanto, las siguientes líneas son datos de entrada del programa (sin considerar los espacios sobrantes):

Castro José A 2 3.751 15  
 Marmolejo Guillermo B 1 1.004 2  
 Estrada Edith E 4 4.000 35

El procedimiento *leerregistros* lee estos registros hasta encontrar un archivo o almacenar cien registros, lo que suceda primero. Los registros se deben indicar, pero no se deben tomar en cuenta. Por cada registro leído, *leerregistro* sumará uno al entero *tamaño* y asignará el valor apropiado a los campos de *alumcomp[tamaño]*. El procedimiento *ponregistros* imprime en proceso y produce una línea aceptable para *leerregistros* por cada elemento del arreglo *alumcomp*.

- 5 Considérense las siguientes declaraciones:

```

TYPE
  tiempo = RECORD
    día: 1..366;      (* día del año *)
    hora: 0..23;      (* hora del día *)
    minuto: 0..59     (* minuto de la hora *)
  END;
  regev = RECORD
    nombre: PACKED ARRAY [1..30] OF char;
    -      tiempoinic, tiempofin: tiempo
  END;
VAR
  evento: ARRAY [1..100] OF regev;
  lapsos: ARRAY [1..2] OF tiempo;

```

Escríbase un segmento de código en Pascal que exhiba el nombre y duración en minutos de todos los sucesos que se inicien y terminen durante el intervalo especificado mediante los tiempos *lapsos[1]* y *lapsos[2]*.

### ★ ★ ★ EJERCICIOS ESTIMULANTES

- 6 Supóngase que una sociedad genealógica tiene un arreglo de registros que dicen la fecha de nacimiento y subíndice en el arreglo de los registros padres y madres de varias personas. El arreglo de registros y las variables correspondientes podrían ser parecidos a éstos:

TYPE

```

máxpersonas = 1000;
unombre = PACKED ARRAY [1..50] OF char;
unapersona = RECORD
    nombre: unombre;
    fechnac: RECORD (* fecha de nacimiento *)
        mes: 1..12;
        día: 1..31;
        año: 0..máxint
    END;
    indmadre,
    indpadre: 0..máxpersonas
END;
```

VAR

```

historia: ARRAY [1..máxpersonas] OF unapersona;
usted: unombre;
```

Los campos *indmadre* e *indpadre* contienen el subíndice en el arreglo *historia* de los registros de la madre y el padre. Este valor será cero si no se dispone de la información correspondiente. Supóngase que la variable *usted* contiene el nombre de una persona; exhiban entonces los nombres y fechas de nacimiento de los padres y los cuatro abuelos de esa persona.

## PROBLEMAS DEL CAPÍTULO 11 PARA RESOLUCIÓN EN COMPUTADORA

### ★ PROBLEMAS ESENCIALES

- 1 Cada una de las líneas de datos de entrada contiene nombres que tienen dos partes, un nombre de pila y un apellido. Ninguna de las partes puede ser mayor de 20 caracteres. Pueden ir precedidas o seguidas de espacios en blanco y estarán separadas mediante uno o más espacios. El ejemplo de datos de entrada ilustra el formato. Escribase un programa en Pascal que lea los datos de entrada y produzca una lista en orden alfabético de los nombres, ordenada primero por apellido y después por nombre de pila.

Ejemplo de entrada:

Juan Pérez

Francisco Pérez

Francisco Martínez

Tseng-Ching Wang Eugenia Pérez

Cresencio Soberanes

Ejemplo de salida:

Francisco Martínez

Eugenia Pérez

Francisco Pérez  
 Juan Pérez  
 Cresencio Soberanes  
 Tseng-Ching Wang

- 2 Cada una de las líneas de datos de entrada para este problema contiene un nombre, dirección, tipo de transacción e importe de la transacción en un sistema de facturación sencillo. El nombre y la dirección serán de 20 caracteres cada uno. El tipo de la transacción puede ser *C* (crédito) o *D* (débito). El importe de la transacción se da en centavos. Si se supone que el saldo inicial de una persona es cero, un crédito reduce el saldo y un débito lo aumenta, escribase un programa en Pascal que produzca un informe, ordenado por nombre, en el que se indique el saldo final en dólares y centavos de cada cuenta. Si un saldo es exactamente cero, el saldo de esa persona no deberá aparecer en el informe.

Ejemplo de entrada:

Federico Estrada	Calle Cinco	# 213	D	4103
Susi Jones	Ave. Libertad	# 9102	D	1000
Federico Estrada	Calle Cinco	# 213	C	1
Bruno Díaz	Calle Oriente	# 2000	D	45000
Susi Jones	Ave. Libertad	# 9102	C	550
Jaime García	Cien Robles	# 913	C	2500
Federico Estrada	Calle Cinco	# 213	C	4102

Ejemplo de salida:

Nombre	Dirección	Saldo
Bruno Díaz	Calle Oriente # 2000	450.00
Jaime García	Cien Robles # 913	-25.00
Susi Jones	Ave. Libertad # 9102	4.50

### ★ ★ PROBLEMAS IMPORTANTES

- 3 Este problema es similar al problema 1 en tanto que los datos de entrada tienen exactamente el mismo formato. La salida, empero, deberá incluir los apellidos en orden alfabético, uno por renglón, seguidos en el mismo renglón por una lista, separada por comas y ordenada alfabéticamente, de los nombres de pila que corresponden a ese apellido. Si se usan los mismos datos de entrada que se mostraron en el problema 1, la salida deberá ser así:

Mártinez	Francisco
Pérez	Eugenia, Francisco, Juan
Soberanes	Cresencio
Wang	Tseng-Ching

- 4 Escribáse un programa que tenga como datos de entrada otro programa en Pascal. El objetivo de este programa es producir una lista de todos los procedimientos y funciones incluidos en el programa, seguida de una lista de los números de línea donde se hizo referencia a los procedimientos y funciones. Puede suponerse que no habrá más de diez procedimientos y funciones y que no se hace referencia más de diez veces a cada uno. Supóngase además que los nombres de los procedimientos y funciones no pasan de quince caracteres. Considérese que las funciones o procedimientos se definen siempre mediante la inclusión de los identificadores FUNCTION o PROCEDURE, y que no hay "trucos" en los datos de entrada. Es decir, no habrá líneas como

(\* PROCEDURE falsalarma; \*)

Ejemplo de entrada:

[Cualquier programa en Pascal]

Ejemplo de salida (representativa):

<u>Nombre</u>	<u>Tipo</u>	<u>Invocaciones</u>
Leedatos	Función	75, 120, 215
Escribe	Procedimiento	90, 95, 96
Clasifica	Procedimiento	201
Iniciales	Procedimiento	195

### ★ ★ ★ PROBLEMAS ESTIMULANTES

- 5 Un departamento de policía está desarrollando un sistema para validar las coordenadas proporcionadas por sospechosos en sus investigaciones. Una de las partes del sistema es un arreglo de registros, cada uno de los cuales tiene la siguiente estructura:

RECORD

```
nombressosp: unombre;
tieneocoart: Boolean
nomcoartada: unombre;
```

END

donde *unombre* es un arreglo empacado de caracteres. Si un sospechoso no tiene coartada, *tieneocoart* será *false*. En caso contrario *nomcoartada* contendrá el nombre de la persona que proporciona la coartada. Supóngase que el nombre del sospechoso aparece en las primeras 25 columnas de una linea de datos de entrada y, si el sospechoso tiene coartada, que el nombre de la persona que la proporciona aparece después de la columna 25. Escribáse un programa en Pascal para leer todos los datos de sospechosos y coartadas y producir dos informes. El primero será una lista de todos los individuos que no

tengan coartada. El segundo informe identificará grupos de personas que proporcionen coartadas para otros del mismo grupo (por ejemplo, A es la *coartada* de B, B es la *coartada* de C y C es la *coartada* de A). Supóngase que no hay más de 50 sospechosos.

Ejemplo de entrada:

Federico Dedos	José Transa
José Transa	Pedro el Matasiete
María Miserias	Federico Dedos
Irma Inocencia	
Pedro el Matasiete	José Transa

Ejemplo de salida:

**Sospechosos sin coartada**

1. Irma Inocencia

**Sospechosos con coartadas circulares:**

1. Pedro el Matasiete
2. José Transa

# CAPÍTULO 12

## CAPÍTULO 12 ARCHIVOS

Tipos y  
variables  
de archivo

Archivos  
de texto

Resolución  
de  
problemas

# ARCHIVOS

## OBJETIVOS

Después de completar este capítulo, el lector deberá ser capaz de:

- Definir tipos de archivo
- Reconocer y declarar variables de archivo
- Reconocer y aplicar las rutinas estándar de manipulación de archivos:*reset*, *rewrite*, *get* y *put*
- Escribir programas para procesar archivos de texto
- Resolver, probar y depurar programas que usen archivos

## PANORAMA GENERAL DEL CAPÍTULO

En este capítulo se estudiará otro tipo de datos estructurado, el archivo. Un **archivo** es una secuencia ordenada de componentes del mismo tipo. Los archivos son diferentes de los arreglos en cuanto a que no se especifica su tamaño, y su longitud puede ser arbitraria. Como se verá, los archivos también se distinguen de los arreglos por la forma de acceso.

Los archivos se guardan normalmente en dispositivos de almacenamiento secundario como son los discos y cintas magnéticos. Es posible pasar la identificación de un archivo como parámetro al programa completo en Pascal al especificar parámetros en el encabezado del programa. Estos archivos se llaman **archivos externos** y, como tales, tienen una duración independiente de los programas que los manipulan. A esto se debe que sean útiles para almacenar permanentemente datos y programas. Los archivos que no aparecen como parámetros en el encabezado del programa son **archivos internos** y existen únicamente durante el tiempo de ejecución del programa; su contenido se pierde al terminar el programa.

En la primera sección de este capítulo se analiza la definición de archivos y la declaración de variables de archivo. Todos los archivos están asociados a una variable de almacenamiento temporal (buffer) de archivo; dicha variable se estudiará en este capítulo. También se analiza el procesamiento de archivos. Pascal cuenta con cinco rutinas estándar para manipulación de archivos: *reset*, *rewrite*, *get*, *put* y *eof*. Se aplicarán éstas para resolver el problema de copiar un archivo.

En la sección 12.2 se examina el tipo de archivo que se usa con más frecuencia, el archivo de texto. Este tipo de archivo se caracteriza por líneas formadas de caracteres, cada una de las cuales termina con un carácter de fin de línea. El lector ya está familiarizado con los procedimientos adicionales para manipulación de archivos de texto, *read*, *readln*, *write* y *writeln*. La función *eoln* se usa también en el procesamiento de archivos de texto. En esta sección se presentan aplicaciones que incluyen el procesamiento de archivos de texto.

La sección de resolución de problemas analiza una operación muy importante que se realiza con archivos, **la fusión**, en la que se produce un solo archivo ordenado mediante la combinación de dos o más archivos

ordenados. Se analizará un algoritmo detallado y se aplicará a un problema de archivos específico. La sección de técnicas de prueba y depuración presenta algunos errores comunes que se pueden presentar si no se usan correctamente los archivos.

## SECCIÓN 12.1 TIPOS Y VARIABLES DE ARCHIVOS

En ocasiones se deseará almacenar información permanente en dispositivos de almacenamiento secundario. Las razones pueden ser muchas, pero en general se busca que los datos estén disponibles para que otros programas los procesen, quizás en sistemas de cómputo físicamente diferentes, o se desea almacenar los datos con el fin de poder continuar su procesamiento en una fecha posterior. Por ejemplo, supóngase que una empresa grande tiene un archivo de 5000 registros de empleados almacenado en cinta magnética. Para tener acceso a la información de este archivo, es preciso sujetarse a las siguientes limitantes:

- 1 Puesto que los registros están almacenados de manera secuencial en el archivo, para tener acceso al registro  $N$  es menester leer el primer registro, el segundo y así sucesivamente hasta el  $N - 1$  antes de tener acceso al registro  $N$ . Esto es parecido a la forma como se puede tener acceso a una melodía determinada grabada en un casete de audio junto con otras melodías. Es preciso pasar todas las melodías que están grabadas en la cinta antes de la que interesa en ese momento.
- 2 Si el archivo tiene un gran número de registros, es posible que no se puedan mantener en la memoria primaria de la computadora todos los registros al mismo tiempo. Por tanto, puede ser necesario procesar solamente una fracción de los registros a la vez, a menudo un solo registro.

Para realizar este tipo de procesamiento de información, Pascal cuenta con un tipo de datos estructurado llamado *archivo (file)*. Puede decirse que un archivo es una secuencia de componentes, todos del mismo tipo. No existe un límite previamente establecido del número de componentes que se pueden almacenar en un archivo (aunque las características físicas del almacenamiento secundario del sistema de cómputo pueden de hecho dar pie a un límite). Estos archivos se llaman también *archivos secuenciales*, ya que el acceso a los componentes debe observar una secuencia estricta. Es decir, al iniciar el procesamiento del archivo es necesario procesar el primer componente del archivo. El siguiente componente al que se puede tener acceso es el segundo y así sucesivamente, hasta procesar todos los componentes deseados.

Sólo algunos sistemas y lenguajes permiten el procesamiento no secuencial de archivos; algunas características no estándar del Pascal pueden también permitir el acceso a un componente determinado de un archivo sin leer todos los componentes que le preceden. Estos archivos se llaman de *acceso aleatorio* o *acceso directo*. No se estudiará este tipo de procesamiento, dado que las características no estándar de Pascal que se requieren para llevarlo a cabo varían en los diferentes sistemas. Aunque estas características no son estándar, pueden ser útiles, y se re-

comienda al lector familiarizarse con ellas si están disponibles en su sistema. Si opta por utilizarlas en sus programas, debe estar preparado para invertir el tiempo necesario para modificar el programa si lo transfiere a un sistema de cómputo diferente.

## Tipos de archivo

La definición del tipo estructurado de archivo se parece a la de un arreglo. Sin embargo, al declarar un arreglo es preciso especificar (indirectamente) su tamaño al dar el tipo del índice. Los archivos, en cambio, no tienen un tamaño fijado previamente. Un archivo puede tener cualquier número de componentes, de cero en adelante. En un arreglo, es posible tener acceso directo a cualquier componente y para ello basta especificar el nombre del arreglo y el valor apropiado del índice. No sucede así con los componentes de un archivo. En un momento dado sólo se dispone de un componente del archivo (llamado *ventana del archivo*). No es posible tener acceso inmediato a los demás componentes del archivo, como serían el que precede o sigue a la ventana del archivo. Es menester mover la ventana al componente deseado.

La definición de tipo de un archivo tiene la siguiente forma:

**TYPE**

identificador = FILE OF tipo de los componentes;

Éstos son ejemplos de definiciones de tipo de archivos:

**TYPE**

archnúm = FILE OF integer;

libro = FILE OF char;

lotería = FILE OF ARRAY [1..1.,5] OF integer;

Obsérvese que el identificador de archivo va seguido de un signo de igual y las palabras reservadas FILE OF, seguidas a su vez de la especificación de tipo de los componentes. El tipo de los componentes puede ser cualquiera, simple o estructurado, excepto el de archivo o cualquier tipo que tenga componentes de tipo archivo. Así, no se permite un archivo que tenga archivos como componentes, ya sea en forma directa o indirecta.

Pascal incluye un tipo estructurado previamente definido llamado *text* (*texto*) que sirve para declarar archivos de texto. Por ejemplo, en todo este libro se han empleado los archivos estándar *input* y *output*. Estos archivos se declaran automáticamente como de tipo *text*. Es decir, el compilador realiza automáticamente la siguiente declaración:

**VAR**

input, output: text;

Después de los archivos de texto, el tipo de archivos más común es el de registros. Un ejemplo de ello sería el archivo de registros de los empleados de una empresa. Los registros se podrían definir así:

**TYPE**

```
empleado = RECORD
  nombre: PACKED ARRAY [1..30] OF char;
  dirección: PACKED ARRAY [1..50] OF char;
  númss: PACKED ARRAY [1..11] OF char;
  Salario: real;
  exenciones: 0..máxint
END;
```

Ahora ya se puede declarar una variable de archivo para los registros de los empleados:

```
VAR archpers : FILE OF empleado;
```

También se podría definir un tipo de archivo y después emplear ese tipo en la declaración de una variable de archivo:

**TYPE**

```
archemp = FILE OF empleado;
```

**VAR**

```
archeprs: archemp;
```

**Parámetros de archivos**

Las variables de archivo *input* y *output* se declaran previamente y son del tipo *text* (del que se hablará en la siguiente sección); todas las demás variables de archivo se deben declarar explícitamente. Cualquier variable de archivo, ya sea *input*, *output* o declarada globalmente en el programa, se puede incluir como parámetro en el encabezado del programa. Estos archivos se llaman **archivos externos**. Tienen una duración mayor que el tiempo de ejecución del programa que los crea o los usa y cuentan con nombres mediante los cuales se puede hacer referencia a ellos fuera del ámbito del programa en Pascal. Se pueden manipular mediante los programas de utilería de que dispone el sistema de cómputo (para copiarlos, borrarlos, editarlos, cambiarles de nombre, etc.) y programas escritos en otros lenguajes de programación. No obstante, en lo que toca al programa en Pascal, la única diferencia entre los archivos externos y las variables de archivo *internas* es que los nombres de variable de los archivos externos deben aparecer en el encabezado en el programa, pero los nombres de variable de archivos internos no deben incluirse. En casi todas las versiones de Pascal, los únicos nombres que aparecen como **parámetros de programa** son los nombres de variables de archivo; Pascal no define la forma en que se asocian estos parámetros con valores. Es frecuente que el nombre de la variable de archivo se asocie explícitamente con el nombre

(fuera de Pascal) que tiene el archivo. Examíñese, por ejemplo, la siguiente proposición:

**PROGRAM** actualizar (*input*, *output*, *basedatos*);

Normalmente el sistema de cómputo contará con un mecanismo para asociar a *input*, *output* y *basedatos* con nombres de archivos. Un sistema IBM, por ejemplo, podría emplear los comandos

```
//OUTPUT DD SYSOUT = A
//BASEDATOS DD DSN = DATOSEN85
//INPUT DD *
      líneas para introducir las variables de archivo
/*
//EXEC ACTUALIZA
```

o los comandos DEC VMS

```
$ DEFINE BASEDATOS DATOSEN85
$ RUN ACTUALIZA
```

o el comando CDC

```
$ ACTUALIZA(, DATOSEN85)
```

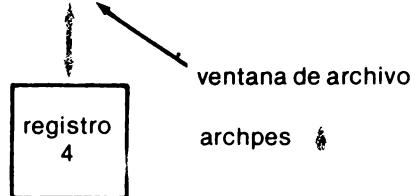
o algunos otros comandos para solicitar la asociación de archivos externos con las variables de archivo en Pascal y la ejecución del programa. Se recomienda al lector examinar su documentación con el fin de averiguar la forma exacta de asociar los nombres de archivos externos con las variables de archivo de Pascal.

### Almacenamiento temporal para archivos (*buffers*)

Al declarar una variable de archivo se crea automáticamente una ventana de archivo que permitirá examinar los componentes del archivo. Esta ventana se llama formalmente *variable de almacenamiento temporal (buffer) del archivo*. El nombre de esta variable es el nombre de la variable de archivo con una flecha ( $\uparrow$ ) o acento circunflejo ( $\wedge$ ) a su derecha. (La mayor parte de los sistemas no cuentan con la flecha, por lo que se emplea el acento circunflejo. No obstante, con objeto de hacer más claro este texto se usará aquí la flecha.) Por ejemplo, *input* $\uparrow$ , *output* $\uparrow$  y *archpers* $\uparrow$  son nombres de variables de almacenamiento temporal de archivos. Todas tienen el mismo tipo que los componentes del archivo correspondiente. En el caso de *input* y *output*, el tipo de la variable de almacenamiento temporal de archivo es *char*, aunque el tipo de estos archivos es *text*. El tipo de *archpers* $\uparrow$  es *empleado*. La figura 12-1 muestra un ejemplo de la variable de almacenamiento temporal de archivo y su relación con el archivo. Obsérvese que en la variable de almacenamiento temporal de archivo (*archpers* $\uparrow$ ) hay una copia del valor que se encuentra actualmente en la ventana del archivo (registro 4). La variable de almac-



... .



Variable de almacenamiento temporal

**Figura 12-1 Variable de almacenamiento temporal de archivo.**

Almacenamiento temporal es el enlace entre el programa y los componentes del archivo. Recuérdese que esta variable se crea siempre que se declara una variable de archivo, de manera que las declaraciones

**TYPE**

```
archnúm = FILE OF integer;
archjuego = FILE OF ARRAY [1..3, 1..3] OF char;
```

**VAR**

```
números: archnúm;
juegos: archjuego;
```

ocasionarán la creación de dos variables de almacenamiento temporal de archivo, un entero *números* ↑ y un arreglo *juegos* ↑.

Estas variables de almacenamiento temporal son como cualquier otra variable, y se pueden manipular de la misma manera. Por ejemplo, las siguientes declaraciones que abarcan variables de almacenamiento temporal de archivo son todas válidas:

<i>índice</i> := <i>número</i> ↑;	(* asignar la variable de buffer a índice *)
<i>número</i> ↑ := 5;	(* asignar cinco a la variable de buffer *)
writeln( <i>archpers</i> ↑. nombre)	(* exhibir nombre de un empleado *)

### Procedimientos y funciones estándar para manipulación de archivos

Pascal incluye los procedimientos estándar *reset*, *rewrite*, *get* y *put* y la función estándar *eof*. En conjunto, éstos permiten la manipulación de variables de archivo. Todas éstas se estudiarán de manera detallada.

En primer lugar se considerará los pasos necesarios para recuperar información de un archivo ya existente. Antes de usar la variable de almacenamiento temporal de un archivo es preciso preparar el archivo para la lectura. Esta “apertura” del archi-

vo se logra mediante el procedimiento *reset*. Si *archent* es la variable de archivo asociada con el archivo que se desea usar, entonces

### **reset (archent)**

establecerá la conexión necesaria entre la variable de archivo y el archivo. También prepara el archivo para leerlo (o “inspeccionarlo”) y coloca la ventana de manera que el primer componente del archivo esté en la variable de almacenamiento temporal *archent*. La figura 12-2 ilustra la situación que prevalece después de haberse ejecutado el procedimiento *reset*. Es *imprescindible* invocar al procedimiento *reset* antes de poder comenzar la lectura. Basta con hacer la invocación una sola vez, y en el caso del archivo estándar *input*, esto se hace automáticamente si su nombre aparece en el encabezado del programa. Si se desea escribir un programa que no utilice el archivo de entrada estándar, no es necesario incluirlo en el encabezado como parámetro del programa.

De manera similar, si se va a grabar (o “generar”) un archivo, es preciso invocar el procedimiento *rewrite* (reescribir). Por ejemplo, si *archsal* es una variable de archivo, entonces

### **rewrite (archsal)**

preparará al archivo correspondiente para ser grabado. Al igual que *reset*, *rewrite* establece la conexión entre la variable de archivo y el archivo. Pero *rewrite* “abre” un archivo para efectuar una salida, por lo que el contenido anterior del archivo se borra totalmente y la ventana queda en el principio del archivo, sobre un componente cuyo valor no está definido.

Una vez preparado un archivo para lectura o grabación, el primer componente del archivo estará disponible en la variable de almacenamiento temporal. En el caso de un archivo de entrada, es posible utilizar inmediatamente el primer componente si así se desea. La proposición

### **get (archent)**

**Figura 12-2 Reset.**



moverá la ventana del archivo al siguiente componente de *archent*, cuyo contenido estará entonces disponible en la variable de almacenamiento temporal *archent*<sup>†</sup>. Si no se puede lograr esto porque se ha llegado al final del archivo y no quedan más componentes, se presentará un error.

La grabación en un archivo es similar en tanto que la variable de almacenamiento temporal es el medio de comunicación de los valores. El valor que va a grabarse se asigna a la variable de almacenamiento temporal y después se invoca el procedimiento *put*. Por ejemplo, supóngase que *nuevodato* es una variable de registro del tipo *empleado* que se definió antes. Para grabar este valor en el siguiente componente de *archnuevo* (archivo de tipo *empleado*) se escribirá

```
archinuevo †:= nuevodato;
put (archnvevo)
```

Al igual que en el caso de los archivos de entrada, es preciso invocar el procedimiento *rewrite* antes del primer intento de invocar el procedimiento *put* con ese archivo.

La función *eof* sirve, como siempre, para determinar si quedan más componentes en un archivo de entrada. Se usó antes con el archivo de texto estándar *input*, pero se puede usar con archivos cuyos componentes sean de cualquier tipo. Por ejemplo, la expresión

```
eof (archent)
```

tendrá el valor *true* si no existen más componentes en el archivo asociado con la variable de archivo *archent*. Esta condición también implica que el valor de la variable de almacenamiento temporal *archent*<sup>†</sup> queda sin definir, por lo que no se debe usar. Aunque normalmente no sirve de mucho, *eof* siempre tendrá el valor *true* si la variable de archivo que es su argumento se está grabando (es decir, está en modo de “generación”). Recuérdese que *eof* sin parámetro se maneja como *eof (input)*.

Ahora se estudiará un problema cuya solución se puede realizar con estos procedimientos.

### Problema 12.1

*Escríbase un procedimiento llamado copiarchivo que produzca un duplicado de un archivo de registros de tipo estudiante, los cuales de definen así:*

```
TYPE
  estudiante = RECORD
    nombre: PACKED ARRAY [1..30] OF char;
    ident: 0..máxint;
    prom: real;
    año: (primero, segundo, tercero, cuarto)
  END;
```

La solución a este problema requiere un programa que tenga dos variables de archivo, una para la entrada (*archent*) y otra para la salida (*aechsal*). La solución se muestra en seguida. Tómese nota de la forma de usar las variables de almacenamiento temporal.

```

PROGRAM copiarchivo (archent, archsal);
(* Copiar un archivo de registros de alumnos de archent a archsal. *)
TYPE
  estudiante = RECORD
    nombre: PACKED ARRAY [1..30] OF char;
    ident: 0..máxint;
    prom: real;
    año: (primero, segundo, tercero, cuarto)
  END;
VAR
  archent, archsal: FILE OF estudiante;
BEGIN
  reset (archent); (* abrir para leer *)
  rewrite (archsal); (* abrir para grabar *)
  WHILE NOT eof(archent) DO
    BEGIN
      archsal := archent; (* copiar registro en archivo de salida *)
      put (archsal); (* grabar componente en archivo de salida *)
      get (archent) (* avanzar al siguiente componente de archent *)
    END
  END.

```

Obsérvese que las proposiciones del programa no dependen realmente del tipo de archivo que se va a copiar. Si se cambia la declaración del tipo *estudiante* se podrá copiar casi cualquier tipo de archivo. La única excepción es el tipo de archivo *text*. En la siguiente sección se analizará la solución al problema de duplicar archivos de texto.

### Variables de archivo

Las variables de archivo, como *archpers* y *archent*, se semejan a las demás variables en que se tienen valores susceptibles de modificación. No obstante, los valores que poseen estas variables son secuencias de componentes, por lo que sólo se pueden manipular mediante los procedimientos y funciones predefinidas de Pascal. Conviene tener presentes los siguientes puntos:

- 1 Las variables de archivo se pueden usar como parámetros verdaderos al invocar procedimientos y funciones, pero el parámetro formal correspondiente debe ser siempre un parámetro variable. Por ejemplo, un procedimiento que requiera una variable de archivo de tipo *tipoarch* comenzaría con la proposición

```
PROCEDURE procesearch (VAR t: tipoarch);
```

- 2 Las variables de archivo no pueden formar parte de una proposición de asignación, aun cuando los tipos de las variables empleadas sean idénticos. Así, los archivos *a1* y *a2*, ambos de tipo *tipoarch*, no pueden aparecer en una proposición de asignación como ésta:

*a1* := *a2*;

La modificación y el acceso a los componentes de un archivo sólo se pueden realizar componente por componente mediante los procedimientos predeclarados con que cuenta Pascal.

- 3 Las variables de archivo no se pueden abrir simultáneamente para lectura y grabación. Si un archivo está abierto para lectura, no se deben invocar los procedimientos *write*, *writ ln* y *put* para manipular ese archivo. Igualmente, si un archivo está abierto para grabación, no se deben invocar procedimientos *read*, *readln* y *get* para ese archivo. De hacerlo, resultará un error.

## EJERCICIOS DE LA SECCIÓN 12.1

- 1 Determíñese cuáles de las siguientes definiciones de tipo de archivo son válidas:

- a) TYPE archmate = FILE OF real;
- b) TYPE file = FILE OF char;
- c) TYPE superarch =FILE OF file;
- d) TYPE mientrada = text;
- e) TYPE bolsa = FILE OF ARRAY [1..10] OF integer;
- f) TYPE secreto = FILE OF text;

- 2 Examínese la siguiente declaración de archivo:

```
TYPE
    archdenum = FILE OF integer;
    archmatriz = FILE OF ARRAY [1..100] OF real;
VAR
    numeros: archdenum;
    datcient: archmatriz;
```

Determíñese cuáles de las siguientes proposiciones son válidas:

- a) numeros ↑ := 5;
- b) datcient[2] ↑: 2;
- c) numeros ↑ := datcient [1];
- d) datcient ↑:= numeros ;
- e) datcient↑[100] := numeros ↑ \* datcient↑[100];

## 3 Encuéntrense dos errores en el siguiente programa:

```

PROGRAMA error (archent, archsal);
BEGIN
    rewrite (archsal);
    WHILE NOT eof (archsal) DO
        BEGIN
            archsal ↑:= archent↑;
            put(archsal);
            get(archsal)
        END
    END.

```

- 4 Escribase un procedimiento que copie todos los números positivos de un archivo de números reales en otro archivo de números reales, sin copiar los números no positivos.
- 5 Escribase un procedimiento que convierta un archivo de reales en un archivo de enteros mediante la sustitución de cada número real por el entero más cercano a él.
- 6 Explíquese el error que tiene cada una de las siguientes proposiciones. Supóngase que se hizo esta declaración:

TYPE

```

    nombre = FILE OF char;
    número = FILE OF integer;

```

VAR

```

    archnom : nombre;
    archnúm : número;

```

- a) PROCEDURE archcomp (archnom: nombre);
- b) archnom := archnúm;
- c) archnom ↑:= archnúm↑;
- d) rewrite (archnom);
- e) get (archnom);
- f) reset (archnúm);
- g) put (archnúm);

## SECCIÓN 12.2 ARCHIVOS DE TEXTO

Recuérdese que los archivos estándar *input* y *output* son archivos de texto. Los archivos de texto son especiales en tanto que están formados por una secuencia de líneas de tamaño variable, cada una de las cuales contiene un número variable de caracteres seguidos de un carácter especial de fin de línea. Los programas en Pascal no pueden grabar caracteres de fin de línea (de manera normal) por lo que debe usarse el procedimiento estándar *writeln*. Además, el carácter de fin de línea no se puede detectar durante la lectura de un archivo de texto si no es mediante la función estándar *eoln*. Estas diferencias implican que los archivos de texto no son

simplemente archivos de tipo *char*, los cuales se pueden considerar como secuencias de caracteres de longitud arbitraria.

Los archivos de texto son los de uso más frecuente porque los caracteres están organizados en líneas. Corresponden a la forma usual de pensar de las personas en lo que se refiere a archivos de material escrito. Al leer este libro, el lector examina líneas, cada una de las cuales contiene un número variable de caracteres. Al escribir una carta las palabras se organizan en líneas.

Los procedimientos estándar *get* y *put* se pueden emplear para leer y grabar archivos de texto. Puesto que las líneas de estos archivos contienen a menudo representaciones de constantes enteras y reales, se cuenta con los procedimientos *read*, *readln*, *write* y *writeln* para que su lectura y escritura sea más sencilla. Ya se han usado todos estos procedimientos en capítulos anteriores (llamándolos proposiciones). Ahora se estudiarán con mayor detalle.

Considérese las declaraciones

#### VAR

```
archpal: text;
uncar: char;
```

Los procedimientos *read* y *write* con *archpal* y *uncar* tienen equivalentes en términos de *get* y *put*. La siguiente tabla ilustra esos equivalentes.

<i>llamadas de procedimiento read/write</i>	<i>equivalente con get/put</i>
<i>write (archpal, uncar)</i>	<i>archpal ! := uncar;</i> <i>put (archpal)</i>
<i>read (archpal, uncar)</i>	<i>uncar := archpal !;</i> <i>get (archpal)</i>

El lector recordará, por haberlo usado antes, que lo último que se hace durante la invocación del procedimiento *writeln* es la grabación de un carácter de fin de línea. Si se invoca el procedimiento *writeln* sin parámetros, o sólo con un parámetro de archivo, escribirá únicamente un carácter de fin de línea.

Recuérdese también que lo último que se hace durante la invocación del procedimiento *readln* es pasar por alto el siguiente carácter de fin de línea y todos los caracteres que le preceden. Si se invoca a *readln* sin parámetros, o sólo con un parámetro de archivo, pasará por alto todos los caracteres hasta el siguiente carácter de fin de línea, inclusive.

Los procedimientos *read*, *readln*, *write* y *writeln* pueden tener varios parámetros cuando se usan con archivos de texto. Tales invocaciones son equivalentes a una secuencia de invocaciones, cada una de las cuales incluye un solo parámetro y la misma variable de archivo (si se usó). Por ejemplo, la proposición

```
read (archpal, car1, car2, car3)
```

es equivalente a

```
read (archpal, car1);
read (archpal, car2);
read (archpal, car3);
```

pero la proposición

```
readln archpal, (archpal, car1, car2, car3)
```

es equivalente a

```
read (archpal, car1);
read (archpal, car2);
readln (archpal, car3)
```

puesto que el salto a la siguiente línea se hace únicamente al final de la invocación de procedimiento. Si se omite la variable de archivo al invocar un procedimiento *read* o *readln*, o una función *eof* o *eoln*, se usará el archivo estándar *input*. De manera similar, si se omite la variable de archivo en una invocación de procedimiento *write* o *writeln*, se usará el archivo estándar *output*.

En la última sección se hizo notar que la variable de almacenamiento temporal de un archivo tiene el mismo tipo que los componentes de ese archivo. Normalmente, las variables y expresiones que se empleen con los procedimientos *read*, *readln*, *write* y *writeln* deberán ser también del tipo adecuado. En el caso de los archivos de texto, eso no es necesario. Los parámetros pueden ser de tipo entero, real o de caracteres, y las invocaciones de *write* o *writeln* pueden incluir también parámetros de tipo booleano y arreglos empacados de caracteres.

Cuando se encuentra un parámetro entero o real en una invocación de procedimiento *read* o *readln*, se espera que el archivo de texto que se lee incluya una secuencia de caracteres que representen una constante entera o real, posiblemente precedida de cualquier número de espacios en blanco o caracteres de fin de línea. El procedimiento efectúa automáticamente la conversión entre esta secuencia de caracteres y la forma interna de un número entera o uno real. Igualmente, al encontrarse un parámetro cuyo tipo no sea *char* en la invocación de procedimiento *write* o *writeln*, se grabará en el archivo de texto la secuencia de caracteres que representen una constante del mismo tipo, posiblemente ajustada por las instrucciones de formato que se incluyan después del parámetro. Por ejemplo,

```
write (output, 17:4)
```

hará que se graben en el archivo de texto *output* dos caracteres de espacio en blanco seguidos de los caracteres ‘1’ y ‘7’.

La función *eoln(a)* adquiere el valor *true* si el componente que se encuentra actualmente en la variable de almacenamiento temporal del archivo *a* (es decir, *a*<sup>↑</sup>) es el carácter de fin de línea. El lector podría pensar, pues, que se puede copiar el carácter de fin de línea en una variable de carácter si se escribe algo parecido a esto:

```
uncar := a↑
```

cuando *eoln(a)* vale *true*. Es posible escribir una proposición así, pero la función *eoln* es la única forma de detectar que *a*↑ contiene un fin de línea. Cualquier otro uso que se le dé a *a*↑ producirá un espacio en blanco (' '). No existe una constante de caracteres que corresponda al carácter de fin de línea; por tanto, no es posible escribirlo sin utilizar el procedimiento estándar *writeln*.

Ahora ya se puede estudiar el problema de duplicación de un archivo de texto. La solución estará organizada de manera similar a como están organizados los archivos de texto.

**PROGRAM copiatexto (archen, archsal);**

**(\* Duplicar un archivo de texto \*)**

**VAR**

    uncar: char;

    archent, archsal: text;

**BEGIN**

    reset (archent);

    rewrite (archsal);

**WHILE NOT eof (archent) DO**

**BEGIN**

**WHILE NOT eoln (archent) DO**

**BEGIN**

                    read (archent, uncar);

                    write (archsal, uncar)

**END;**

                readln (archent);

                writeln (archsal)

**END**

**END.**

## Edición de archivos de texto

Como un ejemplo adicional de procesamiento de archivos de texto, se supondrá que se desea modificar el contenido de un archivo de texto. Por ejemplo, supóngase que se desea eliminar todas las palabras mal escritas de un archivo. Puesto que no es posible modificar el archivo de entrada, se podría copiar este archivo en un archivo interno, haciendo las modificaciones conforme se copia el texto, cambiar a modo de grabación sobre el archivo original y copiar en él el archivo interno. Esta técnica es un poco peligrosa, ya que una interrupción de la energía u otro problema del equipo de cómputo durante la grabación del archivo original podría provocar un desastre. No obstante, se aplicará aquí la técnica a un problema real.

### Problema 12.2

*Escríbase un programa que elimine todas las proposiciones de depuración, cuya forma es*

**(\* DEPURA \*) write... ;**

de un programa fuente en Pascal. Supóngase que los caracteres (\* DEPURA \*) aparecen siempre al principio de estos renglones y que van seguidos, en la misma línea, por la proposición de depuración que se debe eliminar.

El problema se resuelve mediante el siguiente programa. Tómese nota de que los dos procedimientos *leelínea* y *grabalínea* simplifican considerablemente la lectura y grabación de líneas de un archivo de texto. Dado que el archivo de texto usado se proporciona como parámetro, se pueden utilizar tanto con el archivo de entrada como con el de salida. Una función llamada *depurar* determina si la línea actual comienza con el indicador de depuración.

---

```

PROGRAM quitadepur (progpascalent, progpascalsal);
(* Quitar líneas de depuración de un programa fuente en Pascal. *)
CONST
    lonmáxlin = 150;          (* longitud máxima de las líneas *)
    indic = '(* DEPURA *)'    (*indicador de líneas de depuración *)
    lonindic = 12;            (* longitud del indicador de depuración *)
TYPE
    línea = PACKED ARRAY [1..lonmáxlin] OF char;
    cadena = PACKED ARRAY [1..lonindic] OF char;
VAR
    progpascalent,           (* programa original *)
    progpascalsal: text;     (* programa sin líneas de depuración *)
    líneactual: línea;
    longactual: integer;
PROCEDURE leelínea (VAR archent: text; VAR línent: línea;
                     VAR lonlínea: integer);
(* leer una línea del archivo de texto archent y colocarla en línent, *)
(* colocar en lonlínea el número de caracteres de la línea.      *)
(* eof (archent) debe ser false al invocar leelínea.          *)
VAR
    c: char;
BEGIN
    lonlínea := 0;           (* inicialmente no hay caracteres *)
    WHILE NOT eoln(archent) DO
        BEGIN
            read (archent, c);      (* capturar siguiente carácter *)
            IF lonlínea < lonmalin
            THEN BEGIN             (* almacenarlo si hay espacio *)
                lonlínea := lonlínea + 1;
                línent[lonlínea] := c
            END
        END
        readln (archent)          (* pasar por alto fin de línea *)
    END; (* de leelínea *)

```

```

PROCEDURE grabalínea (VAR archsal: text; línsal: línea;
    tamlín: integer);
(* grabar línsal[1..tamlín] en el archivo archsal *)
(* y agregar un carácter de fin de línea      *)
VAR
    i : integer;
BEGIN
    FOR i := 1 TO tamlín DO          (* grabar los caracteres *)
        write (archsal, línsal[1]);
        writeln (archsal);           (* grabar el carácter de fin de línea *)
END; (* de grabalínea*)
FUNCTION depurar (VAR unalín: línea; tamlín: integer;
    indic: cadena): booleana;
(* true si unalín comienza con el indicador de depuración *)
(* false en caso contrario *)
VAR
    temp: boolean;
    i: 1..lonindic;
BEGIN
    IF tamlín < longind          (* ¿es demasiado corta la línea? *)
        THEN depurar := false
        ELSE BEGIN
            temp := true;
            FOR i := 1 TO tamindic DO (* ver si hay indicador *)
                temp := temp AND (unalín[1] = indic[1]);
                depurar := temp
            END
        END
    END;
(* Programa principal *)
BEGIN
    reset (progpascalent);
    rewrite (progpascalsal);
    WHILE NOT eof (progpascalent) DO
        BEGIN
            leelínea (progpascalent, líneactual, longactual);
            IF NOT depurar (líneactual, longactual, indic)
                THEN grabalínea (progpascalsal, líneactual, longactual)
        END
    END.

```

Obsérvese que la entrada de este programa es un programa en Pascal, pero de hecho podría ser cualquier archivo de texto. La salida sería una copia exacta de la entrada sin las líneas que comiencen con (\* DEPURA \*). Por ejemplo, considérese los siguientes datos de entrada.

PROGRAM muestra (output):

```

VAR i: integer;
BEGIN
    FOR i := 1 TO 10 DO
(* DEPURA *) BEGIN
(* DEPURA *)     writeln (Variable i = , i:2);
(* DEPURA *)     writeln ('Hola')
(* DEPURA *) END
END.

```

Después de la ejecución, el archivo de salida tendría las siguientes líneas:

```

PROGRAM muestra (output);
VAR i: integer;
BEGIN
    FOR i := 1 TO 10 DO
        writeln ('Hola')
END.

```

Este programa se puede modificar fácilmente con el objeto de realizar varias transformaciones en las líneas de un archivo de texto mediante el copiado de un archivo a otro. Por ejemplo, la conversión de todas las letras minúsculas en mayúsculas se podría lograr con una modificación sencilla. Este tipo de programas se llaman *filtros*.

## EJERCICIOS DE LA SECCIÓN 12.2

- 1 Supóngase que *frases* es un archivo de texto que contiene estos datos:

```

Ser o no ser.<eoln>
Esto sobre todo:<eoln >
Sé fiel a ti mismo <eoln >
El resto es silencio.<eoln >
<eof >

```

Determíñese la salida del siguiente segmento en Pascal (supóngase que *frases* es un archivo de texto, *incóg* un entero y *uncar* una variable de carácter):

```

reset (frases);
incóg := 0;
WHILE NOT eof (frases) DO
BEGIN
    WHILE NOT eoln (frases) DO
        read (frases, uncar);

```

```

incóg := incóg + 1;
readln (frases)
END;
writeln (incóg)

```

- 2 ¿Puede el lector determinar si en el problema 1 la variable *incóg* está relacionada con el contenido del archivo del texto? Si es así, determíñese la salida si *frases* se declara como

**FILE OF char;**

- 3 Escríbase un procedimiento que lea un archivo de texto en el que la longitud de las líneas varíe entre uno y 72 caracteres y que cree otro archivo de texto en el que todas las líneas tengan una longitud de exactamente 72 caracteres mediante el relleno (al final) de cada línea con cero o más espacios en blanco.
- 4 Escríbase un procedimiento en Pascal que lea una línea de un archivo de texto como datos de entrada y exhiba el número de caracteres que no son espacios en cada línea.
- 5 Escríbase un procedimiento en Pascal que lea una línea de un archivo de texto y la almacene en un arreglo empacado. Si la línea tiene menos de 72 caracteres, rellénese con espacios. Supóngase que se definió globalmente el tipo del arreglo empacado:

**TYPE**

```
cadena = PACKED ARRAY [1..72] OF char;
```

## SECCIÓN 12.3 RESOLUCIÓN DE PROBLEMAS CON ARCHIVOS

Durante el procesamiento de archivos, una operación muy solicitada es la fusión de dos archivos. Un *archivo fusionado* es un solo archivo clasificado que se produce mediante la combinación de dos o más archivos clasificados. Considérese el siguiente problema.

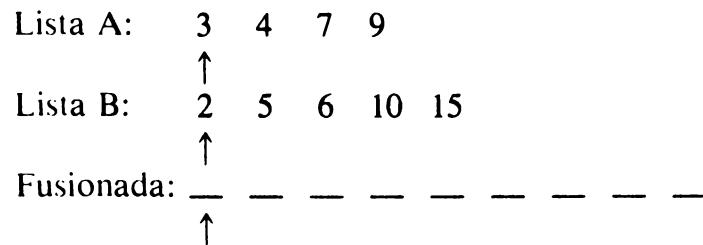
### Problema 12.3

*Las divisiones de equipo y aplicaciones de Electrónica Apex se van a combinar en una sola división de sistemas. Los registros de personal de ambas divisiones se almacenan actualmente en archivos separados en los que los registros están clasificados en orden ascendente por número de seguro social. Escríbase un programa que fuse los dos archivos de registros de personal para producir el archivo de personal de la división de sistemas. Este nuevo archivo debe estar también clasificado en orden ascendente por número de seguro social.*

El problema principal aquí radica en la fusión. Antes de intentar resolver en detalle este problema, se estudiará un ejemplo de fusión más sencillo que emplea

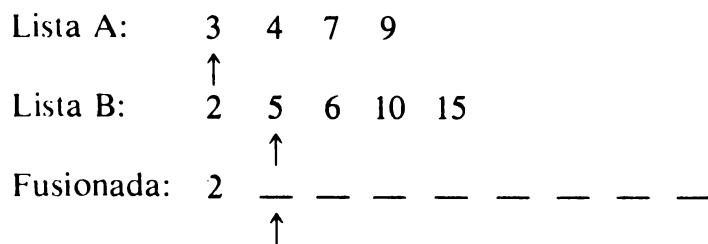
## Fusión

Supóngase que se tienen dos listas de enteros clasificados en orden ascendente y una lista fusionada vacía, como se muestra en seguida. Obsérvese que las listas pueden tener tamaños diferentes.



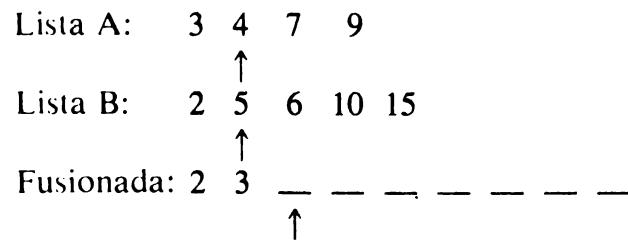
Cada una de las listas incluye un indicador ( $\uparrow$ ) que apunta al número más pequeño de la lista. Para fusionar las dos listas se hace lo siguiente. Se examina cada una de las listas de menor a mayor, conservando los valores de las posiciones actuales (según señalan los indicadores). Las posiciones actuales se comparan y se inserta el valor más pequeño en la posición actual de la lista fusionada. Los indicadores que apuntan al valor más pequeño y a la posición actual en la lista fusionada se avanzan un elemento a la derecha, mientras que la tercera flecha (que apunta al valor que no se insertó) no se mueve. Este proceso se repite hasta que una de las listas queda vacía, momento en el que se agregan los valores restantes de la lista que no está vacía a la lista fusionada.

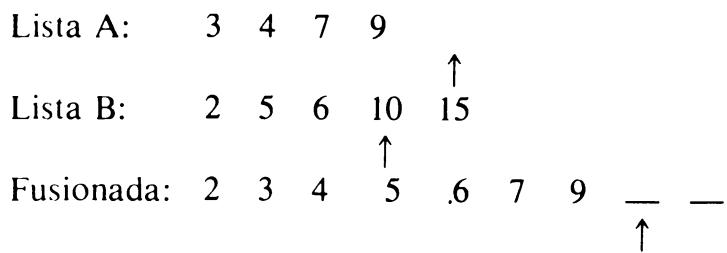
Volviendo al ejemplo, después de la primera comparación e inserción las listas tienen el siguiente aspecto:



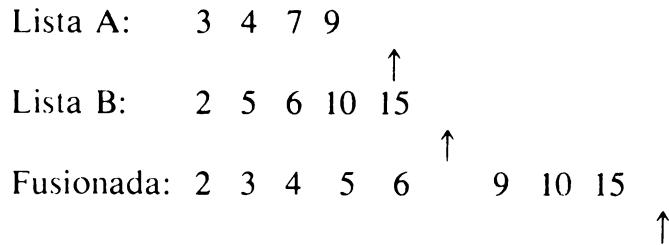
Obsérvese que el indicador de posición actual se avanzó en la lista B, ya que el valor anterior fue el más pequeño. El indicador de posición de la lista fusionada siempre se avanza, ya que siempre se inserta un valor.

Al repetir el proceso se ve que la siguiente iteración produce los siguientes cambios:





El último paso es copiar los valores restantes de la lista B (todos los cuales deben ser mayores que cualquier valor de lista A) en la lista fusionada. Una vez hecho esto, los indicadores quedarán así:



Ahora ya es posible bosquejar el algoritmo de fusión. Supóngase que se van a fusionar los arreglos *listaA* y *listaB* para producir el arreglo *listaC*. Los indicadores de posición se llaman *posiciónA*, *posiciónB* y *posiciónC*.

#### *Algoritmo de fusión*

```

Asignar valores iniciales a posiciónA, posiciónB y posiciónC.
Mientras (no fin de listaA) y (no fin de listaB)
    Si listaA[posiciónA] < listaB[posiciónB],
        Entonces listaC[posiciónC] = listaA[posiciónA].
        Incrementar posiciónA.
    Si no, listaC[posiciónC] = listaB[posiciónB].
        Incrementar posiciónB.
        Incrementar posiciónC.
    Si fin de listaA,
        Entonces copiar resto de listaB en listaC.
    Si no, copiar resto de listaA en listaC.

```

## Resolución del problema

Ahora que se bosquejó una solución al problema de la fusión, se puede pasar a la resolución del problema original. Supóngase que los registros de empleados que tenían las divisiones de equipo y aplicaciones de Electrónica Apex están descritos por esta definición TYPE:

## TYPE

```

    empleado = RECORD
        númss: PACKED ARRAY [1..11] OF char;
        nombre: PACKED ARRAY [1..30] OF char;
        salario: real;
        exenciones: 0..máxint
    END
    archemp = FILE OF empleado;

```

Las variables de archivo se pueden declarar entonces así:

## VAR

```

    aplicaciones,
    equipo,
    sistemas: archemp;

```

Se utilizará un procedimiento llamado *fusarch* basado en el algoritmo de fusión antes presentado con una modificación apropiada a los tipos de registros que se van a combinar. Por ejemplo, el tipo *archivo* en Pascal crea automáticamente los indicadores de posición del algoritmo, y lleva la cuenta del número de componentes procesados en cada archivo. A pesar de ello, podría ser conveniente mantener una cuenta del número de registros procesados, en cuyo caso se usarían los indicadores de posición. La solución del problema es la siguiente:

---

```

PROGRAMA fusión (aplicaciones, equipo, sistemas);
(* Combinar los archivos de personal de las divisiones de *)
(* aplicaciones y equipo para producir el archivo de      *)
(* personal de la división de sistemas                  *)
TYPE
    empleado = RECORD
        númss: PACKED ARRAY [1..11] OF char;
        nombre: PACKED ARRAY [1..30] OF char;
        salario: real;
        exenciones: 0..máxint
    END;
    archemp = FILE OF empleado;

VAR
    aplicaciones,
    equipo,
    sistemas: archemp;
PROCEDURE fusarch (VAR ent1, ent2, sal: archemp);
(* fusionar los archivos ent1 y ent2 para producir el archivo *)
(* sal en orden de número de seguro social ascendente      *)

```

```

BEGIN
  WHILE NOT eof(ent1) AND NOT eof(ent2) DO
    BEGIN
      IF ent1↑.númss < ent2↑.númss
      THEN BEGIN
        sal↑ := ent1↑;
        get (ent1)
      END
    ELSE BEGIN
      sal↑ := ent2↑;
      get (ent2)
    END;
    put (sal)
  END
(* copiar resto de los registros en archivo sal *)
WHILE NOT eof(ent1) DO
BEGIN
  sal↑ := ent1↑;
  put (sal);
  get (ent1)
END
WHILE NOT eof (ent2) DO
BEGIN
  sal↑ := ent2↑;
  put (sal);
  get (ent2)
END
END; (* de fusarch *)
(* programa principal *)
BEGIN
  reset (aplicaciones);
  reset (equipo);
  rewrite (sistemas);
  fusarch (aplicaciones, equipo, sistemas)
END.

```

---

## SECCIÓN 12.4 TÉCNICAS DE PRUEBA Y DEPURACIÓN

En esta sección se examinarán algunos errores comunes relacionados con el procesamiento de archivos. Un error frecuente se debe a la confusión entre los procedimientos *reset* y *rewrite*. Si se omite ya sea la invocación de *reset* o de *rewrite* en el programa, es probable que no se presenten errores de compilación, pero es seguro que el programa fallará al ejecutarse.

Una vez aplicado el procedimiento *reset* a un archivo, las únicas operaciones que se pueden aplicar a ese archivo son *read*, *get* y *readln* si el tipo del archivo es

*text*. De manera similar, una vez aplicado el procedimiento *rewrite* a un archivo, las únicas operaciones que se deben usar son *write*, *put* y *writeln* si el tipo del archivo es *text*. Un archivo se puede usar tanto para entrada como para salida, siempre que se usen *reset* y *rewrite* en la forma adecuada y no se trate de leer y grabar al mismo tiempo. Recuérdese, empero, que tan pronto como se invoca el procedimiento *rewrite* se pierde el contenido anterior del archivo.

### Errores de *eof* y *eoln*

Es probable que los errores más frecuentes en el manejo de archivo se presenten con las funciones *eof* y *eoln*. Supóngase que se tiene un archivo de texto llamado *archent* cuyos últimos componentes son los siguientes:

```
5 3_ 2 1 _ _ <eoln> <eof>
```

Considérese el siguiente código (suponiendo que *núm* se declara como entero):

```
WHILE NOT eof(archent) DO
BEGIN
  read (archent, núm);
  write ('El número es', núm)
END
```

Este ciclo dará como resultado un error cuando se ejecute debido a un intento de leer más allá del fin de archivo. El problema es que, tan pronto como se lee el entero 21, el siguiente carácter no es el <eof>, sino un espacio en blanco. Aunque se hubieran omitido los últimos espacios, todavía habría problemas, ya que <eoln> no “disparará” la función *eof*.

Para evitar este error conviene ejecutar una proposición *readln* antes de una prueba de fin de archivo. Puesto que el fin de archivo sólo puede presentarse después de un fin de línea, esto garantizará que la prueba se realice en el lugar correcto.

Otro problema similar se puede presentar al leer más de un elemento en una sola invocación de los procedimientos *read* o *readln*. Supóngase que se usa la proposición

```
read (archent, dato1, dato2, dato3)
```

pero se encuentra un fin de archivo antes de capturar con éxito un valor para *dato3*. El sistema de cómputo informará también en este caso de un intento de leer más allá del fin de archivo. Para evitar estos problemas siempre se debe ejecutar un *readln* antes de *eof* y se tiene un número variable de datos (ya sea intencional o accidentalmente) leerlos uno por uno, de preferencia carácter por carácter en el caso de archivos de texto.

La función *eoln* puede ser fuente de muchos errores. Recuérdese que *eoln* sólo se puede usar con archivos de texto, aunque *eof* se puede emplear con cualquier tipo de archivo. No se olvide que el carácter de fin de línea es efectivamente un espacio en blanco cuando lo lee el programa y que la función *eoln* es la única forma

de detectar su presencia en un archivo de entrada. He aquí un segmento de código diseñado para copiar *archent* en *archsal* (ambos son del tipo *text* y *uncar* es de tipo *char*):

```
reset (archent);
rewrite (archsal);
WHILE NOT eof(archent) DO
BEGIN
  read (archent, uncar);
  write (archsal, uncar);
  IF eoln (archent)
  THEN writeln (archsal)
(* ERROR -- No se ha leido el fin de linea *)
END;
```

En este ejemplo faltó leer el carácter de fin de línea (mediante *readl*), cuando se le detectó. A resultas de esto se agregará un espacio en blanco adicional a todas las líneas de datos de entrada con excepción de la primera.

Al aplicar *reset* a un archivo que está vacío, la condición *eof* se cumple inmediatamente, de manera que si el archivo *archent* está vacío, el siguiente código provocará un error:

```
reset (archent);
REPEAT
  read (archent, uncar);
(* ERROR -- faltó verificar si el archivo está vacío. *)
  write (archsal, uncar)
UNTIL eof(archent);
```

Para evitar este problema, conviene verificar siempre si se llegó al fin de archivo antes de procesar ese archivo.

He aquí una lista de recordatorios importantes de Pascal en el aspecto de archivos.

## RECORDATORIOS DE PASCAL

- *Eoln, readln y writeln* no se pueden usar con archivos que no sean de texto.
- Al usar *read* y *write* con archivos que no sean de texto, sólo se puede leer o grabar un componente a la vez.
- *Rewrite* abre un archivo para grabarlo. El contenido anterior del archivo se pierde.
- *Reset* abre un archivo para lectura; si el archivo está vacío, *eof* vale *true* inmediatamente.
- El procedimiento *get* mueve la ventana del archivo y asigna el siguiente componente del archivo a la variable de almacenamiento temporal correspondiente.
- El procedimiento *put* agrega el valor de la variable de almacenamiento temporal del archivo al final del archivo.
- Las variables de archivo no se pueden emplear en asignaciones.
- Las variables de archivo sólo se pueden usar como parámetros de procedimientos y funciones que esperan parámetros variables.
- Todas las variables de archivo, con excepción de *input* y *output*, se deben declarar.

**SECCIÓN 12.5 REPASO DEL CAPÍTULO**

En este capítulo se estudiaron los archivos y su procesamiento. En Pascal, un archivo es un tipo de datos estructurado que contiene una secuencia de componentes, todos del mismo tipo. Los archivos no tienen un límite de tamaño (excepto por limitaciones físicas). Los archivos externos pueden usarse para almacenar datos que tengan una vida independiente de la ejecución del programa. Se analizó la definición y declaración de archivos y variables de archivo. También se presentó la variable de almacenamiento temporal (*buffer*) de archivo. Se analizaron los procedimientos *reset*, *rewrite*, *get* y *put* y la función *eof*, aplicándolos a un problema de duplicación de archivos.

En la sección 12.2 se estudiaron los archivos de texto y el uso de los procedimientos y funciones de Pascal *read*, *readln*, *write*, *writeln* y *eoln*. Se presentó una aplicación de manejo de archivos (edición de un archivo).

En la sección 12.3 se analizó la fusión de dos archivos. Se desarrolló un algoritmo de fusión y se modificó para manejar archivos.

En seguida se presenta el resumen de las características de Pascal analizadas en este capítulo ya que puede servir como referencia en el futuro.

**REFERENCIAS DE PASCAL**

- 1** Un archivo es una secuencia de componentes, todos del mismo tipo. La forma general de las definiciones de tipo archivo es:

**TYPE**

identificador = FILE OF tipo de los componentes;

Ejemplo:

**TYPE**

archcar = FILE OF char;

- 2** Rutinas de manejo de archivos.

- 2.1 *Reset (archent)* abre archivo el *archent* para lectura y asigna el primer componente a la variable de almacenamiento temporal del archivo (*archent*↑).
- 2.2 *Rewrite (archsal)* abre el archivo *archsal* para grabación, y borra el contenido que se pudiera haber tenido.
- 2.3 *Get (archent)* asigna el siguiente componente de *archent* a la variable de almacenamiento temporal (*arch ent*↑).
- 2.4 *Put (archsal)* agrega el contenido actual de la variable de almacenamiento temporal (*archsal*↑) al archivo.
- 2.5 *Eof (archent)* vale *true* si la ventana del archivo está colocada más allá del último componente de *archent*.
- 2.6 *Eoln (archent)* sólo se permite si *archent* es un archivo de texto y la variable de almacenamiento temporal (*archen*↑) contiene un carácter de fin de línea.

**3 Archivos de texto:**

- 3.1 Tipo de archivo predefinido.
- 3.2 Organizado en forma de líneas, cada una de las cuales contienen un número variable de caracteres seguidos de un carácter de fin de línea.
- 3.3 Los componentes más elementales son caracteres.
- 3.4 Es posible leer números enteros y reales de archivo de texto con *read* y *readln*.
- 3.5 Es posible grabar valores enteros, reales y booleanos en archivos de texto con *write* y *writeln*.
- 3.6 *Eoln* sirve para detectar la presencia de un carácter de fin de línea.

**4 Declaraciones de archivo.**

- 4.1 Los archivos externos aparecen como parámetros del programa.

**PROGRAM muestra (archent, archsal);**  
**VAR archent, archsal: FILE OF algúntipo;**

- 4.2 Los archivos internos no aparecen como parámetros del programa.

## Avance del capítulo 13

En el siguiente capítulo se concluirá el análisis de los tipos de datos estructurados con el tipo de conjunto. También se presentará el tipo de datos de apuntador con aplicaciones a otras estructuras de datos como son las listas encadenadas y los árboles.

## Palabras clave del capítulo 12

archivo de acceso directo	fusión
<i>eof</i>	<i>put</i>
<i>eoln</i>	archivo de acceso aleatorio
archivo externo	<i>read</i>
archivo	<i>readln</i>
variable de almacenamiento	<i>reset</i>
temporal de archivo	<i>rewrite</i>
parámetro de archivo	archivo secuencial
ventana de archivo	<i>text</i>
filtros	<i>write</i>
<i>get</i>	<i>writeln</i>
archivo interno	

## EJERCICIOS DEL CAPÍTULO 12

### \* EJERCICIOS ESENCIALES

- I En muchos sistemas de cómputo, los caracteres de fin de línea se representan realmente mediante un sólo carácter en los archivos de texto. Por ello, un

archivo que se pueda leer como archivo de texto también se puede leer como un archivo de caracteres (FILE OF *char*), con ciertas limitaciones. Una de ellas es que no se puede usar la función *eoln* con la variable de archivo.

- a) ¿Cuál otra restricción se aplica a la lectura de un archivo de caracteres?
- b) Supóngase que una variable global de carácter llamada *findelinea* contiene el carácter que representa al fin de línea en archivos de texto. Escríbase una función booleana llamada *ceoln* con un parámetro variable de archivo de tipo FILE OF *char* que tenga la misma función que la función *eoln*.

- 2 Aunque es muy ineficiente, es posible tener acceso a un archivo de Pascal en forma directa simulada. Si se aplicó el procedimiento *reset* a un archivo *a*, por ejemplo, una segunda invocación de *reset* simplemente volverá a ajustar el archivo de manera que la variable de almacenamiento temporal contenga el primer componente. El siguiente procedimiento, por ejemplo, leerá en forma “directa” el componente *i* del archivo *a* (que se supone será del tipo *tipoarch*), y asignará el registro elegido a la variable de almacenamiento temporal.

```
PROCEDURE accesodirecto (VAR a: tipoarch; i integer);
BEGIN
  reset (a);
  WHILE (i > 1) AND NOT eof (a) DO
    BEGIN
      get (a);
      i := i - 1
    END
  END;
```

Explíquese el funcionamiento de este procedimiento. En particular, explíquese qué hay en la variable de almacenamiento temporal cuando el valor que se da para *i* es mayor que el número de componentes en el archivo. ¿Qué sucede si *i* es uno y el archivo está vacío (es decir, no tiene componentes)?

### ★ ★ EJERCICIOS IMPORTANTES

- 3 Estúdiese de nuevo el ejercicio 2, pero supóngase que se desea extraer *n* (variable entera) registros del archivo, cuyos números de registro se han almacenado en los componentes *loc[1]* a *loc[n]* de un arreglo de enteros. Supóngase además que los registros extraídos se deben almacenar en *reg[1]* a *reg[l]*, donde *reg* es un arreglo cuyos componentes tienen el mismo tipo que los del archivo. Escríbase un procedimiento en Pascal que extraiga los registros deseados del archivo, invocando *reset* únicamente una vez.

### ★ ★ ★ EJERCICIOS ESTIMULANTES

- 4 El acceso directo a los registros de un archivo para grabación en Pascal es todavía más costoso que el acceso directo para lectura, como se muestra en seguida. Una técnica que se puede usar para simular la grabación por acceso directo del registro *i* de un archivo *a* de tipo FILE OF *tipocomponentes* se resume a continuación:

- a) Grabar (con *rewrite*) un archivo interno *temp* del mismo tipo que *a*.
- b) Copiar los registros de *a* desde el primero hasta *i* — 1 en *temp*.
- c) Grabar los datos nevos del registro *i* en *temp*.
- d) Pasar por alto el registro *i* de *a* (es decir, leerlo y asignarlo a una variable de “desperdicio” del tipo *tipocomponentes*).
- e) Copiar el resto de los registros de *a* a *temp*.
- f) Por último copiar *temp* en *a*.

Escríbase un procedimiento en Pascal llamado *grabadirecto* cuyos parámetros sean una variable de archivo, una posición de registro entera y un registro que se debe grabar en esa posición. No debe olvidarse la condición excepcional de que la posición del registro especificado sea mayor que el número de registros que contiene actualmente el archivo.

## EJERCICIOS DEL CAPÍTULO 12 PARA RESOLUCIÓN DE COMPUTADORA

### ★ PROBLEMAS ESENCIALES

- 1 El procedimiento estándar *page(a)* (véase el apéndice G) origina una acción definida en la instalación de cada sistema de Pascal. Su objetivo es hacer que la salida al archivo de texto *a* comience en una “página” nueva. Esto puede hacer que se avance el papel de una impresora hasta la siguiente forma, o que se borre la pantalla y se comience a exhibir la salida en la esquina superior izquierda de las terminales de video. Escríbase un programa que use el procedimiento *page* para producir un listado de un archivo de texto con números de línea y 50 líneas por página. Supóngase que los números de línea ocuparán las primeras posiciones de cada línea de salida. Después del número de línea, escribanse dos espacios y una línea del archivo de texto.

Ejemplo de entrada:

Ésta es la línea 1.

Ésta es la línea 2.

...

Ésta es la línea 50.

Ésta es la línea 51.

Ejemplo de salida:

1 Ésta es la línea 1.

2 Ésta es la línea 2.

...

50 Ésta es la línea 50.

<en la parte superior de la siguiente página

51 Ésta es la línea 51.

- 2** El archivo estándar *input* contiene un par de enteros *m* y *n*. Un segundo archivo de texto que no contiene más de 500 líneas, cada una con 80 caracteres o menos, se debe clasificar y grabar en un tercer archivo de texto. El archivo se debe clasificar en orden ascendente según los caracteres de las posiciones *m* a *m* + *n* – 1 de cada línea. Si una línea determinada no tiene por lo menos *m* + *n* – 1 caracteres se deberá llenar con espacios en blanco sólo para determinar su posición en el archivo de salida. Los espacios de relleno no deberán aparecer en el archivo de salida.

### ★ ★ PROBLEMAS IMPORTANTES

- 3** En un archivo de texto *a* aparece una lista de palabras, una por línea. Estas palabras tienen longitudes variables, pero no más de 19 caracteres. Se desea exhibir las palabras en tantas columnas como sea posible, de manera que la anchura total de todas las columnas no sea mayor que un número (constantes) de caracteres que se especifique y todas las columnas queden separadas de las demás por lo menos por un espacio en blanco. Escríbase un programa que realice esta tarea. (Sugerencia: léase una vez todo el archivo *a* para determinar la longitud máxima de las palabras, determinese después el número máximo de columnas que se pueden exhibir y léase de nuevo el archivo de texto; exhibanse las palabras en las columnas.)

Ejemplo de entrada:

Notación polaca  
 Potencial  
 Números primos  
 Entero  
 Etiqueta  
 Letra  
 Fusión  
 Hígado molido  
 Infija  
 Vacío

Ejemplo de salida (se supone una longitud máxima de 55 caracteres):

Notación polaca	Potencial	Números primos
Entero	Etiqueta	Letra
Fusión	Hígado molido	
Vacío		

- 4** Un archivo de texto *p* contiene un prototipo de carta publicitaria. Esta carta no tiene más de 50 líneas, cada una de las cuales contiene un máximo de 50 caracteres. En varios lugares de *p* aparecen (nunca separados al terminar una línea) los caracteres @*n*@, donde *n* es un dígito decimal. Un segundo archivo de texto *m* contiene un número arbitrario de líneas, cada una de las cuales contiene hasta 10 campos de caracteres separados por comas. Para cada línea

de *m*, cópíese el archivo de texto *p* en un archivo de salida *j*, y sustitúyase el campo *n* de la línea de *m* por cada aparición de @*n*@. Si aparece la secuencia de caracteres @*n*@ en *p* tal que *n* es mayor que el número de campos que tiene la línea de actual *m*, simplemente bórrese @*n*@ del texto que se graba en *j*. Sepárense las copias de la carta que se graban en *j* mediante una invocación de *page(j)*.

Ejemplo de archivo *p*:

Estimad @0@ @1@,

Así que adquiriste una computadora @2@ @3@. Bueno @0@, estoy seguro de que quieres aprovechar al máximo tu @3@, de manera que querrás adquirir el sistema Pascal Placentero. Tiene tantas extensiones que no podrás vivir sin él, @0@, Cuesta sólo \$19.95 (no pienses en que gastarás muchos miles de dólares en reescribir tus programas para usarlos en otros sistemas de Pascal.)

Ejemplo de archivo *m*:

o,Carlos,Computófilo,Cháfex  
a,Berta,Binaria,Albatross  
a,Mata,Hari, Crypto 100

Ejemplo de salida en el archivo *j*:

Estimado Carlos Computófilo,

Así que adquiriste una computadora Cháfex. Bueno, Carlos, estoy seguro de que quieres aprovechar al máximo tu Cháfex, de manera que querrás adquirir el sistema Pascal Placentero. Tiene tantas extensiones que no podrás vivir sin él, Carlos. Cuesta sólo \$19.95 (no pienses en que gastarás muchos miles de dólares en reescribir tus programas para usarlos en otros sistemas de Pascal).

< parte superior de la siguiente página >

Estimada Berta Binaria,

Así que adquiriste una computadora Albatross. Bueno, Berta, estoy seguro de que quieres aprovechar al máximo tu Albatross, de manera que querrás adquirir el sistema Pascal Placentero. Tiene tantas extensiones que no podrás vivir sin él, Berta. Cuesta sólo \$19.95 (no pienses en que gastarás muchos miles de dólares en reescribir tus programas para usarlos en otros sistemas de Pascal).

< parte superior de la siguiente página >

Estimada Mata Hari,

Así que adquiriste una computadora Crypto 100. Bueno, Mata, estoy seguro de que quieres aprovechar al máximo tu Crypto 100, de manera que querrás adquirir el sistema Pascal Placentero. Tiene tantas extensiones que no podrás vivir sin él, Mata. Cuesta sólo \$19.95 (no pienses en que gastarás mucho miles de dólares en reescribir tus programas para usarlos en otros sistemas de Pascal).

- 5 Un archivo *ar* es un archivo de texto cuyas líneas tienen la forma

**@nombremódulo**

seguido de un número variable de líneas de texto que forman el contenido de *nombremódulo*. El fin del texto que constituye un módulo se indica mediante el signo **@**, que inicia la siguiente línea, o el fin de archivo. Por ejemplo, un archivo sencillo podría contener

```
@módulo1
Línea 1 del módulo 1
Línea 2 del módulo 1
@módulo2
Línea 1 del módulo 2
@móduloxy15
Línea 1 del módulo xy15
```

Supóngase que un segundo archivo de texto llamado *progent* tiene un programa en Pascal en el que se han omitido módulos que se utilizan con frecuencia (y cuyo texto aparece en el primer archivo). El archivo *progent* contiene dos tipos de líneas, las que comienzan con **@** seguido de un nombre de módulo, y las que no comienzan con **@**. Escribáse un programa en Pascal que copie el archivo de texto *progent* en un tercer archivo de texto llamado *progesal* sin alterar las líneas que no comienzan con **@** y sustituyendo las líneas que comienzan con **@** por las líneas del archivo de texto *ar* que siguen a la línea **@nombremódulo**. Puede suponerse que los nombres de módulo no tienen más de veinte caracteres y que las líneas de todos los archivos de texto usados no contienen más de cien caracteres.

Ejemplo de programa *progent*:

```
PROGRAM muestra (input, output);
@declaraciones
@lector
@escritor
@clasificador
BEGIN
  leecosas;
  clasificacosas;
  escribecosas
END
```

Ejemplo de archivo *ar*:

```
@declareaciones
VAR
```

```
númelem: integer;
cosas: ARRAY [1..1000] OF varios;
@lector
PROCEDURE leecosas;
...
@escritor
PROCEDURE escribecosas;
...
@clasificador
PROCEDURE clasificacosas;
```

Ejemplo de salida en el archivo *progsal*:

```
PROGRAM muestra (input, output);
VAR
    númelem: integer;
    cosas: ARRAY [1..1000] OF varios;
PROCEDURE leecosas;
...
PROCEDURE escribecosas;
...
PROCEDURE clasificacosas;
...
BEGIN
    leecosas;
    clasificacosas;
    escribecosas
END
```



# CAPÍTULO 13

CAPÍTULO 13  
INTRODUCCIÓN A  
LAS ESTRUCTURAS  
DE DATOS

Apunadores

Árboles

Abstracción de  
los datos

Conjuntos

Listas  
encadenadas

Pilas y  
colas

**INTRODUCCIÓN  
A LAS ESTRUCTURAS  
DE DATOS**

Después de completar este capítulo, el lector deberá ser capaz de:

- Reconocer y aplicar el tipo de datos de conjunto y las operaciones de conjuntos unión, intersección y diferencia
- Definir y declarar apuntadores y variables de apuntador
- Crear variables dinámicas
- Crear y aplicar estructuras de datos dinámicas como son las listas encadenadas, árboles binarios, pilas y colas

## PANORAMA GENERAL DEL CAPÍTULO

En este capítulo se presenta una introducción a las estructuras de datos, especialmente las estructuras de datos dinámicas. Los tipos de datos estructurados que incluye Pascal son los arreglos, registros, archivos y conjuntos. Ya se analizaron los arreglos, registros y archivos en capítulos anteriores. En la sección 13.1 se estudiará el tipo de datos de conjunto. Pascal es uno de los pocos lenguajes estructurados de alto nivel que maneja el tipo de datos de conjunto.

Pascal cuenta con otro tipo de datos predefinido llamado **apuntador**. En la sección 13.2 se hablará de los apuntadores. Una variable de apuntador contiene la dirección de una localidad de memoria. Los apuntadores nuevos permiten la creación de **estructuras de datos dinámicas**, las cuales se pueden ampliar mediante la adición de componentes nuevos, o reducir por la eliminación de componentes durante la ejecución del programa. Las estructuras dinámicas de datos no tienen las restricciones de tamaño que acompañan a otras estructuras, como los arreglos. Recuérdese que el tamaño de un arreglo se debe especificar antes de la ejecución del programa. Las estructuras de datos cuyo tamaño no se puede modificar durante la ejecución (como los arreglos) se llaman **estructuras de datos estáticas**.

El resto del capítulo presenta las estructuras de datos dinámicas que se pueden construir mediante el empleo de apuntadores. Estas estructuras de datos se analizan con mayor detalle en cursos de computación avanzada. Se estudiarán las siguientes estructuras de datos: listas encadenadas, árboles binarios, pilas y colas. La aplicación de estructuras dinámicas como éstas permite muchas veces resolver de manera eficiente y efectiva gran número de problemas complejos de manejo de datos. Como se verá, en el caso de un lenguaje estructurado de alto nivel, como Pascal, es fácil crear este tipo de estructuras de datos dinámicas.

## SECCIÓN 13.1 CONJUNTOS

Pascal incluye un tipo de datos estructurados llamado conjunto. En el capítulo 8 se puede encontrar una introducción a los conjuntos. Un **conjunto** es un grupo de objetos del mismo tipo. Por ejemplo, el conjunto de los números enteros positivos.

[2, 4, 6, 8, 10, 12]

Recuérdese que esta lista se llama *constructor del conjunto*. Los objetos que pertenecen al conjunto se denominan *elementos*. Así, en este ejemplo, 2, 4, 6, 8, 10 y 12 son elementos del conjunto dado. Los elementos deben ser distintos y su orden no es importante; por ejemplo, el siguiente conjunto es igual al que se definió anteriormente:

[12, 10, 8, 6, 4, 2]

Para declarar una variable de conjunto es preciso definir primero el tipo de conjunto. Estúdiese, por ejemplo, la siguiente definición y declaración:

**TYPE**

    número = SET OF 1.50;

**VAR**

    conjnum: número;

La proposición **TYPE** define el conjunto de miembros potenciales, de manera que un conjunto de tipo *número* puede incluir cualquier grupo de enteros escogidos de la escala de 1 a 50, incluso el *conjunto vacío*, [ ]. Es posible valerse de una proposición de asignación para colocar elementos en un conjunto. La siguiente tabla muestra varias asignaciones a la variable de conjunto *conjnum* del ejemplo anterior:

asignación	elementos del conjunto
conjnum : = {1,2,9,10}	1,2,9,10
conjnum : = [1,3,5,7,9]	1,3,5,7,9
conjnum : = [2,4,6,8,10,12]	2,4,6,8,10,12
conjnum : = [10,20,30,40,50]	10,20,30,40,50
conjnum : = [1]	1
conjnum : = [ ]	Ninguno (conjunto vacío)

Los tipos de conjunto tienen la siguiente forma:

**TYPE**

    identificador = SET OF tipo base

El identificador de tipo de conjunto va seguido del signo de igual acostumbrado, de las palabras reservadas SET y OF (conjunto de) y por último del tipo base que describe el tipo de los miembros que se permiten en el conjunto. El tipo base puede ser un tipo simple o enumerado, incluso tipos de subescala.

La mayor parte de las versiones de Pascal imponen una restricción sobre el número máximo de elementos que puede contener un conjunto. Por ejemplo, un sis-

tema requiere que el tipo base no especifique más de 64 valores. En este caso, la definición de tipo de conjunto

**TYPE**

```
número = SET OF 1..128;
```

no se permitiría, ya que el tipo *1..128* permite 128 valores diferentes. De manera similar, la definición de tipo de conjunto

**TYPE**

```
alfa = SET OF char;
```

podría no permitirse si existen más de 64 caracteres distintos. Además, algunos sistemas también exigen que el valor ordinal más pequeño de cualquier miembro de un conjunto no sea negativo. La definición de tipo de conjunto

**TYPE**

```
número = SET OF -50..-1
```

no se permitiría en estos sistemas. El lector debe averiguar si existen restricciones de este tipo en su sistema.

El conjunto que consta de todos los valores posibles del tipo base se llama *conjunto universal*. La siguiente proposición asignará el conjunto universal a la variable de conjunto *conjnúm* que se declaró anteriormente:

```
conjnúm : = [1..50]
```

Los conjuntos contenidos dentro de otro conjunto se denominan *subconjuntos* de ese conjunto. En estos casos todos los elementos del primer conjunto son también elementos del segundo. Por ejemplo, dado el conjunto *[2,4,6,8,10,12]*, los siguientes son subconjuntos de ese conjunto:

```
[2,4],      [6,8,10,12],     [12],      [2,8,12]
```

En seguida se da otro ejemplo de declaraciones de conjunto seguido de asignaciones válidas y no válidas:

**TYPE**

```
conjalfa = SET OF 'A'..'Z';
```

```
materia = (álgebra, historia, computación, inglés);
```

```
cursos = SET OF materia;
```

**VAR**

```
vocales: conjalfa;
```

```
misclases: cursos;
```

**Asignaciones válidas**

```
vocales : = ['A','E','I','O','U']
```

```
vocales : = [ ]
```

```
misclases : = [álgebra, computación]
```

```
misclases : = [álgebra..inglés]
```

vocales : = 'A' (\* el tipo 'A' no es un tipo de conjunto \*)

misclases : = vocales (\* los tipos de conjunto no son idénticos \*)

## Operadores de conjuntos

Existen tres operadores de conjuntos que se pueden usar para combinar dos conjuntos. Tales operadores son unión (+), intersección (\*) y diferencia (-). Supóngase que se declaran las variables *conja* y *conj<sub>b</sub>* como conjuntos del mismo tipo base. En ese caso, la *unión* de *conja* y *conj<sub>b</sub>*

**conja + conj<sub>b</sub>**

es un conjunto cuyos elementos aparecen ya sea en *conja* o en *conj<sub>b</sub>* o en ambos conjuntos. La *intersección* de *conja* y *conj<sub>b</sub>*

**conja \* conj<sub>b</sub>**

es un conjunto cuyos elementos aparecen tanto en *conja* como en *conj<sub>b</sub>*. Por último, la *diferencia* entre *conja* y *conj<sub>b</sub>*

**conja - conj<sub>b</sub>**

es un conjunto cuyos elementos aparecen en *conja* pero no en *conj<sub>b</sub>*. Los siguientes ejemplos deberán aclarar el efecto de estos operadores:

expresión	resultado
[1..2..4..6] + [1..3..5..7]	[1..2..3..4..5..6..7]
[1..2..4..6] * [1..3..5..7]	[1]
[1..2..4..6] - [1..3..5..7]	[2..4..6]
[1..3..5..7] - [1..2..4..6]	[3..5..7]
[2..4..6] * [3..5..7]	[ ]
[1..20] + [16..50]	[1..50]
[1..20] * [16..50]	[16..20]
[1..20] - [16..50]	[1..15]

El orden en que se evalúan la unión, intersección y diferencia de conjuntos es el mismo que el de los operadores aritméticos correspondientes. Es decir, la operación de intersección de conjuntos se efectúa antes que las operaciones de unión o diferencia de conjuntos.

Es posible agregar elementos nuevos a un conjunto mediante el operador de unión, y los elementos existentes pueden suprimirse de un conjunto mediante el operador de diferencia. La proposición

**conj<sub>n</sub>um : = conj<sub>n</sub>um + [10]**

agregará el elemento 10 al conjunto representado por la variable *conjnúm*. Si el elemento 10 ya estuviera en *conjnúm*, este último no se alteraría. De manera similar, la proposición

*conjnúm* : = *conjnúm* – [10]

eliminará el elemento 10 de *conjnúm* si ya existía en él; en caso contrario *conjnúm* no sufre cambio alguno.

Los conjuntos no se pueden leer de un archivo de texto o grabarse en él. En cambio, los elementos de un conjunto se pueden procesar de la forma acostumbrada. Por ejemplo, se pueden leer valores enteros de un archivo de texto de entrada e introducirse al conjunto *conjnúm* de la siguiente forma (supóngase que *núm* es una variable entera):

```
conjnúm : = [ ]; (* asignar valor inicial al conjunto *)
WHILE NOT eof DO
BEGIN
  WHILE NOT eofn DO
    BEGIN
      read (núm); (* leer el siguiente número *)
      conjnúm : = conjnúm + [núm] (* agregarlo al conjunto *)
    END;
    readIn
  END
END
```

La lectura es mucho más sencilla si se procesa un archivo de entrada del tipo de conjunto. Si se supone la siguiente declaración

```
TYPE
  número = SET OF 1..50;
VAR
  conjnúm: número;
  a: FILE OF número
```

entonces, para leer el conjunto *completo*, bastará con escribir

```
reset (a);
read (a, conjnúm)
```

El uso del operador relacional IN permite exhibir el contenido de *conjnúm* en un archivo de texto. Recuérdese del capítulo 8 que es posible probar si un elemento existe en un conjunto mediante el operador IN. Por ejemplo

5 IN [1,2,4,5,6]

es verdadero, pero

3 IN [2,4,6,8]

vale *false*. El siguiente código exhibirá los elementos de *conjnum* (supóngase que *índice* es una variable entera):

```
FOR índice : = 1 TO 50 DO
  IF índice IN conjnúm
    THEN writeln (índice)
```

La grabación de un conjunto en un archivo del tipo apropiado se puede hacer de esta manera (se usa el archivo *a* del ejemplo anterior):

```
rewrite (a);
write (a, conjnúm)
```

También es posible aplicar los operadores relacionales  $=$ ,  $<>$ ,  $<=$ ,  $>=$ , y  $=$ , a los conjuntos. La siguiente tabla define el resultado de aplicar cada uno de los operadores relacionales a una pareja de conjuntos, *conja* y *conjbj*. Obsérvese una vez más que el resultado de los operadores relacionales es booleano.

<i>expresión</i>	<i>significado</i>	<i>ejemplo de valor true</i>
<i>conja</i> = <i>conjbj</i>	Conjuntos iguales	$['A', 'E'] = ['E', 'A']$
<i>conja</i> $<>$ <i>conjbj</i>	Conjuntos diferentes	$[2,4] <> [2,3]$
<i>conja</i> $<=$ <i>conjbj</i>	<i>Conja</i> subconjunto de <i>conjbj</i>	$[1,2,5] <= [1,2,4,5]$
<i>conja</i> $>=$ <i>conjbj</i>	<i>Conjb</i> subconjunto de <i>conja</i>	$['A', 'C'] >= ['A']$

En el siguiente problema se escribirá un procedimiento para calcular la diferencia simétrica de dos conjuntos *a* y *b*. La **diferencia simétrica** de dos conjuntos se define como el conjunto de elementos que están en *a* y no están en *b* junto con el conjunto de elementos que están en *b* y no están en *a*.

### Problema 13.1

*Escríbase un procedimiento llamado simétrica que forme la diferencia simétrica de dos conjuntos a y b cuyo tipo es el siguiente:*

```
TYPE
  alfabeto = SET OF 'A'..'Z';
```

El siguiente procedimiento almacenará la diferencia simétrica en *conjC*:

```
PROCEDURE simétrica (VAR conja, conjb, conjc: alfabeto);
(* calcular la diferencia simétrica de conja y conjb *)
BEGIN
  conjc : = (conja - conjb) + (conjbj - conja)
END;
```

En el siguiente problema se examinará un arreglo de conjuntos.

**Problema 13.2**

*Escribase un programa en Pascal que obtenga y exhiba el conjunto de divisores para cada uno de los enteros entre uno y cien, inclusive. El resultado se debe almacenar en un arreglo de conjuntos llamado número.*

La salida que produce este programa deberá comenzar de esta manera:

Número	Divisor(es)
1	1
2	1 2
3	1 3
4	1 2 4
5	1 5
6	1 2 3 6

El algoritmo empleado para resolver este problema se presenta a continuación.

*Algoritmo de divisores*

Por cada entero llamado *índice* en la escala de 1 a 100:

Asignar valor inicial al conjunto de divisores para que incluya 1.

Por cada número en la escala de 2 a *índice*:

Si el número divide a *índice* sin residuo,

Entonces agregar el número al conjunto de divisores.

Por cada entero llamado *índice* en la escala de 1 a 100:

Por cada entero llamado *númdiv* en la escala de 1 a *índice*:

Si *númdiv* está en el conjunto de divisores de *índice*,

Entonces exhibir *númdiv*.

El programa completo en Pascal se muestra en seguida. Se vale de una técnica de “fuerza bruta” para resolver el problema. En los ejercicios se sugiere una estrategia más eficiente.

---

```

PROGRAM exhibenúm (input, output);
(* Obtener y exhibir el conjunto de divisores para cada *)
(* uno de los números enteros en la escala de 1 a 100 *)
TYPE
  conjdiv = SET OF 1..100;
VAR
  número : ARRAY [1..100] OF conjdiv;
  divisores : conjdiv;
  númdiv, índice : 1..100;
BEGIN
  (* crear el arreglo de divisores *)
  FOR índice : = 1 TO 100 DO
    BEGIN

```

número[índice] : = [1];

(\* verificar si hay otros divisores \*)

FOR númdiv : = 2 TO índice DO

IF índice MOD númdiv = 0

THEN número[índice] : = número[índice] + [númdiv]

END;

(\* exhibir resultados \*)

writeln ('Número Divisor(es)');

writeln ('-----');

FOR índice : = 1 TO 100 DO

BEGIN

write (índice:4,:6);

FOR númdiv : = 1 TO índice DO

IF númdiv IN número[índice]

THEN write (númdiv:1,:1);

writeln (\* saltar a la siguiente línea \*)

END

END.

---

## EJERCICIOS DE LA SECCIÓN 13.1

1 Dígase cuáles de los siguientes tipos de conjunto son válidos:

a) TYPE conjdígitos = SET OF 0..9;

VAR num: conjdígitos;

b) TYPE conjcar = SET OF '0'..'9';

VAR tabla: ARRAY [-10..10] OF conjcar;

c) TYPE arreconj = SET OF ARRAY [-10..10] OF char;

VAR tabla: arreconj;

d) TYPE

marca = (coma, punto, dospuntos, puntoycoma, espacio);

especial = SET OF marca; .

VAR símbolo: especial;

e) TYPE conjcarac = SET OF A..Z;

VAR letra: conjcarac;

2 Dadas las asignaciones de conjuntos

conja : = [1,3,5,7,9,11]

conjB : = [2,4,6,8,10,12]

conjC : = [1,2,3,4,5,7]

determinense los conjuntos que resultan de las siguientes expresiones:

- a) conja + conjb
- b) conja \* conjb
- c) conja + conjc
- d) conjb \* conjc
- e) conja - conjb
- f) conja + conjc \* conjb
- g) (conja - conjb) \* (conjc - conjb)
- h) conja \* conjc \* conjc
- i) conjc - conjc
- j) conja \* (conjc - conjb) + conjc
- k) conja \* conjc - conjb

3 Determíñese el valor de las siguientes expresiones:

- a) [1, 2, 3] = [3, 1, 2]
- b) [12, 14, 16] <= [12, 14, 16]
- c) [3, 6, 7] = [3, 6, 7] \* [1, 3, 6, 7]
- d) [1..20] >= [1..30]
- e) 'B' IN ['C'..'T']

4 Dado  $\text{conja} = [10]$ ,  $\text{conj}b = [1,3,10]$ ,  $\text{conj}c = [0,1,2,3,4]$  y  $\text{conj}d = [3,4,5,6]$ , determíñese el resultado de estas expresiones:

- a)  $\text{conja} * \text{conj}b + \text{conj}c - \text{conj}d$
- b)  $(\text{conj}d - \text{conj}b) - (\text{conj}c * \text{conja})$
- c)  $(\text{conja} + \text{conj}b) \leq \text{conj}c$
- d)  $(\text{conj}c * \text{conj}d) \leq \text{conj}b$
- e) 3 IN  $\text{conj}b * \text{conj}c * \text{conj}d$

5 Examínense estas declaraciones:

**TYPE**

```
nombre = (John, Paul, George, Ringo);
grupo = SET OF nombre;
VAR combo : grupo;
```

Hágase una lista de todos los conjuntos que se pueden construir a partir del tipo base *nombre* y almacenar en la variable de conjunto *combo*. (Sugerencia: existen 16 conjuntos posibles, incluyendo el conjunto vacío.)

- 6 Escríbase un programa en Pascal que lea dos enunciados en español y exhiba el conjunto de letras que aparecen en alguno de los enunciados pero no en ambos. Supóngase que los enunciados terminan con un punto. (Sugerencia: utilícese el procedimiento *simétrica* del problema 13.1.)
- 7 Escríbase una función en Pascal que dé como resultado el número de miembros en un conjunto del tipo

**TYPE** conjnúm = SET OF 1..128

- 8 Modifíquese el programa *exhibenúm* del problema 13.2 para obtener una solución más eficiente. (Sugerencia: obsérvese que los divisores forman parejas. Por ejemplo, cada uno de los divisores produce otro divisor, a saber, el cociente. Por ejemplo, el número 12 se puede expresar así:

$$12 = 1 * 12 = 2 * 6 = 3 * 4$$

No hace falta considerar los divisores que siguen al tres, por lo que los divisores son 1, 2, 3, 4, 6 y 12. La conclusión es que, dado un número, no hace falta considerar los divisores más allá de la parte entera de la raíz cuadrada del número.)

## SECCIÓN 13.2 APUNTADORES

Ya se ha finalizado el análisis de los tipos de datos estructurados de Pascal (arreglos, registros, archivos y conjuntos). Pascal incluye un tipo de datos adicional llamado *tipo de apuntador*. Recuérdese que el tamaño de los arreglos en Pascal debe declararse y fijarse durante la compilación y antes de que se ejecute el programa. Debido a estas restricciones, los arreglos se conocen como *estructuras de datos estáticas*. No obstante, en ocasiones se desea manipular datos cuyo tamaño puede cambiar durante la ejecución. Las estructuras de datos que se pueden modificar durante la ejecución del programa al expandirse o contraerse sin especificar un tamaño fijo se llaman *estructuras de datos dinámicas*. Como se verá en el resto de este capítulo, es posible construir estructuras dinámicas complejas mediante el enlace de componentes con variables llamadas apuntadores.

En Pascal, un *apuntador* es una variable que apunta o *hace referencia* a una localidad de memoria en la que está almacenado un dato. Recuérdese del capítulo 1 que cada una de las celdas de memoria de una computadora tiene una dirección que puede emplearse para tener acceso a esa celda de memoria. Es posible tener acceso al contenido de esta dirección de memoria y modificarlo a través del apuntador usando la notación que se analiza a continuación.

Los tipos de apuntador se definen así:

### TYPE

númapuntador =  $\uparrow$  integer;

En este ejemplo, *númapuntador* define un apuntador que señala una localidad de memoria que contiene un valor entero. El símbolo especial  $\uparrow$ , o el acento circunflejo ( $\wedge$ ) debe preceder al identificador de tipo para definir un tipo de apuntador.

Ahora ya se pueden declarar las variables de apuntador:

### VAR

apun1, apun2 : númapointer;

Los valores almacenados en estas variables de apuntador no están definidos inicialmente, ya que no se les han asignado valores. Recuérdese que el valor de una variable de apuntador va a ser una dirección de memoria. Estos valores dependen

intrínsecamente del sistema de cómputo específico que se esté usando; como resultado de esto, las formas de manipular variables de apuntador están restringidas. No se pueden comparar dos variables de apuntador mediante los operadores relacionales  $>$ ,  $>=$ ,  $<$ , y  $<=$ ; sólo se permite determinar si dos apuntadores son iguales (es decir, si contienen la misma dirección de memoria) o diferentes. Además, no se permite leer o grabar variables de apuntador en archivos de texto. Es decir, las variables de apuntador no tienen representaciones textuales.

Puesto que las variables de apuntador contienen la dirección de una localidad de memoria, existen varias formas de asignar estas direcciones a una variable de apuntador. La primera de ellas, el procedimiento estándar *new*, obtiene la dirección de una localidad de memoria desocupada del tipo apropiado y se la asigna a una variable de apuntador. Por ejemplo, las proposiciones

```
new (apun1);
new (apun2)
```

obtienen las dos direcciones de localidades de memoria que estén actualmente desocupadas y que resulten para almacenar enteros; después asignan una de éstas a *apun1* y la otra a *apun2*. El proceso de obtener estas direcciones de memoria se denomina *asignación dinámica*. Las direcciones de memoria también se llaman *variables de referencia*. Sólo se puede tener acceso a estas variables enteras a través de las variables de apuntador de la siguiente manera:

```
apun1 ↑ (* variable entera a la que apunta apun1 *)
apun2 ↑ (* variable entera a la que apunta apun2 *)
```

Las siguientes son asignaciones válidas a las variables de referencia:

```
apun1 ↑ := 12; (* almacenar 12 en el entero al que apunta apun1 *)
apun1 ↑ := apun1 ↑ + 4 (*sumar 4 al entero al que apunta apun1 *)
```

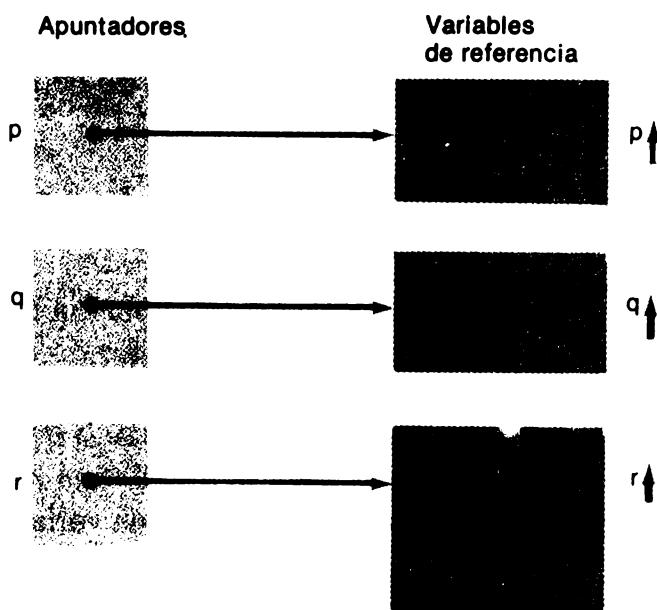
Pero nótese que las proposiciones

```
apun1 ↑ := 'A';
apun1 := 12;
apun1 := apun1 + 4;
```

no son válidas, ya que los tipos de datos de las variables y las constantes incluidas no son compatibles, es decir, no es posible asignar un valor de caracteres a un entero o un valor entero a una variable de apuntador, porque los tipos no son idénticos. Recuérdese que el valor de una variable de apuntador es una dirección que depende del sistema de cómputo específico que se use. La proposición de asignación

```
apun1 := apun2
```

sí se permite, y hace que *apun1* reciba la misma dirección que está almacenada en *apun2*. Después de ejecutar esta proposición de asignación, las variables de referencia *apun1 ↑* y *apun2 ↑* serán idénticas: tanto *apun1* como *apun2* contienen la



**Figura 13-1 Apuntadores y variables de referencia.**

misma dirección de memoria y, por tanto, *apun1* $\uparrow$  y *apun2* $\uparrow$  se refieren al mismo valor entero.

La figura 13-1 ilustra tres variables de apuntador *p*, *q* y *r*, cada una de las cuales apunta a una variable de referencia (dirección de memoria) distinta. Obsérvese que la variable de referencia *p* $\uparrow$  es de tipo entero, *q* $\uparrow$  es de tipo real y *r* $\uparrow$  es de tipo registro.

Pascal incluye también el procedimiento estándar *dispose*, que devuelve una dirección de memoria a la “reserva” de dirección disponibles para asignación mediante *new*. Es decir, *dispose* es el mecanismo empleado para indicar que una dirección de memoria que se asignó previamente *new* ya no se necesita, y para liberarla de manera que se le pueda dar otro uso. Por ejemplo,

**dispose (apun2)**

liberará la dirección de memoria a la que apunta la variable de apuntador *apun2*. Adviértase que todavía existe la variable de apuntador *apun2*, pero su valor no está definido, puesto que ya no hace referencia a una dirección de memoria disponible.

Las variables de apuntador no definidas pueden conducir muchas veces a problemas insospechados. Pascal incluye una constante que se puede asignar a cualquier variable de apuntador para indicar que no apunta a una dirección específica. Esta constante se representa mediante la palabra reservada NIL. La proposición

**apun1 := NIL**

asigna este valor a la variable de apuntador *apun1*. Nótese que una variable de apuntador con el valor NIL es significativamente diferente de una cuyo valor no está definido. Se permite comparar un apuntador que contiene el valor NIL para determinar si es o no igual a otro valor de apuntador, pero nunca se debe usar en

una comparación una variable de apuntador que contenga una dirección no definida; hacerlo causaría un error de ejecución o produciría un resultado no definido. La siguiente proposición ilustra una comparación válida:

```
IF apun1 = NIL
THEN writeln ('Apun1 es nil.')
ELSE writeln ('Apun1 apunta a', apunt1 ^ )
```

Ya se han visto tres técnicas diferentes para cambiar la dirección almacenada en una variable de apuntador:

- 1 Usar el procedimiento estándar *new* para asignar la dirección de una variable de referencia recién asignada.
- 2 Asignar la dirección que está en una variable de apuntador del mismo tipo.
- 3 Asignar el valor NIL.

Recuérdese que las variables de apuntador no se pueden leer de archivos de texto ni exhibirse. Sin embargo, las variables de referencia se pueden manipular de cualquier forma permitida en el caso de variables del mismo tipo. Por ejemplo, *apun1* es una variable de apuntador y, por tanto, existen varias restricciones en cuanto a la forma de manipularla. Por otro lado, *apun1* ^ es una variable entera y se puede manipular como cualquier otra variable entera. Por ejemplo, las siguientes son proposiciones válidas que hacen uso de las variables de apuntador *apun1* y *apun2* recién definidas:

```
read (apunt1 ^ );
writeln (apunt2 ^ );
apun2 ^ := apun1 ^ * apun2 ^ + 3;
IF apun2 ^ < 0
THEN writeln ('La variable de referencia es negativa.')
```

Las variables de apuntador que se han presentado hasta ahora contienen la dirección de variables enteras. Sin embargo, es frecuente que las variables de apuntador señalen a tipos de datos estructurados como son los registros. Estúdiense con sumo cuidado las siguientes definiciones y declaraciones:

```
TYPE
  apuntreg = ^ datosreg;
  datosreg = RECORD
    edad: 21.. 99; (* campos de datos *)
    sexo: (masc, fem);
    salario: real;
    sigreg: apuntreg      (* apuntador *)
  END
VAR
  regpersona: apuntreg;
```

Obsérvese que la definición de tipo de *apuntreg* incluye el tipo de referencia *datosreg* antes de que se haya definido el registro. En Pascal se permite esto en el caso de las definiciones TYPE de apuntadores. Después de ejecutarse la proposición

**new (regpersona)**

el contenido de *regpersona* será la dirección de memoria asignada para el registro. Obsérvese que uno de los campos del registro también es un apuntador de registro. Esto también se permite y, como se verá en la siguiente sección, es posible crear dinámicamente una cadena de registros enlazados entre sí mediante los campos apuntadores.

## EJERCICIOS DE LA SECCIÓN 13.2

- 1 Determíñese cuáles de las siguientes proposiciones son válidas, si se supone que se hicieron las siguientes definiciones y declaraciones:

TYPE

apuntador $\triangleq$   $\uparrow$  integer;  
apuntador $\triangleq$   $\uparrow$  char;

VAR

apun1,apun2:apuntadorla;  
apun3,apun4:apuntadorle;

- a) new (apunt1)
- b) new (apunt1  $\uparrow$ )
- c) apun1 := apun3
- d) apun2  $\uparrow$  := apun2  $\uparrow$  + apunt1  $\uparrow$
- e) apun1 := NIL
- f) apun4  $\uparrow$  := NIL
- g) writeln (apun2, apun3)
- h) read (apun1  $\uparrow$ , apun4  $\uparrow$ )

- 2 Supóngase que se hicieron las declaraciones del ejercicio anterior. Determíñese la salida que produce el siguiente código:

```
new (apun1);
new (apun2);
new (apun3);
apun2 := apun1;
apun1  $\uparrow$  := 2;
apun2  $\uparrow$  := 3;
apun3  $\uparrow$  := 'A';
writeln (apun1  $\uparrow$ , apun2  $\uparrow$ , apun3  $\uparrow$ )
```

- 3 ¿Tiene algún error el siguiente código? Supóngase que *apun1* es una variable de apuntador que hace referencia a una variable entera.

```
new (apun1);
read (apun1 ↑ );
writeln (apun1 ↑ );
dispose (apun1);
writeln (apun1 ↑ ) ←
```

- 4 Supóngase que se hicieron las declaraciones del ejercicio 1. Determinese cuáles de las siguientes proposiciones son válidas:

- apun1* := NIL
- apun2* := new (*apun1*)
- dispose (*apun3*)
- apun3* ↑ := NIL
- apun3* := *apun4* AND (*apun3* = NIL)

- 5 Determinese la salida del siguiente código si se supone que se hicieron las declaraciones del ejercicio 1:

```
new (apun3);
new (apun1);
apun3 ↑ := 'Z';
apun2 := NIL;
apun4 := NIL;
IF (apun3 <> NIL) AND (apun2 = NIL)
THEN writeln ('Código A');
IF apun3 ↑ = 'Z'
THEN writeln ('Código Z')
ELSE writeln ('Código X')
```



### SECCIÓN 13.3 LISTAS ENCADENADAS

Los tipos de apuntadores más comunes son los que apuntan a registros. Como se vio en la última sección, cada registro puede contener por lo menos un campo que consta de un apuntador a otro registro. Dada esta situación, es posible crear variables dinámicas que sean registros con apuntadores a otros registros. Estos apuntadores pueden formar una cadena de registros conocida como *lista encadenada*.

¿Qué caso tiene crear una estructura de datos de este tipo si es posible crear un arreglo de registros sin necesidad de una cadena de apuntadores? Considérese el problema de insertar o eliminar un componente de cualquier arreglo. Por ejemplo, el programa de edición de textos del capítulo 10 incluyó procedimientos para insertar y eliminar líneas. El tiempo requerido para agregar o eliminar líneas en el arreglo de líneas puede ser muy largo porque es preciso desplazar los componentes del arreglo. En cambio, como se verá en esta sección, si los componentes forman una lista encadenada, la inserción y eliminación se puede hacer con gran rapidez si se cambian simplemente los valores de los apuntadores. El uso de listas

encadenadas requiere mayor memoria por la necesidad de almacenar apunadores. En otras palabras, se trata de una compensación; emplear más memoria de manera efectiva puede reducir el tiempo requerido para llevar a cabo la tarea.

Ahora se creará una lista encadenada de registros. Supóngase que se hacen las siguientes definiciones y declaraciones (no se especificarán los campos del registro, con excepción del campo apuntador):

#### TYPE

```

apuntador = ↑ unregistro;
unregistro = RECORD
    campo: tipo;      (* campo(s) de datos *)
    sigreg: apuntador
END;
  
```

#### VAR

```

apuntinic: apuntador;
apuntactual: apuntador;
índice: integer;
  
```

Obsérvese que se declararon dos variables de apuntador. *Apuntinic* contendrá la dirección del comienzo de la lista y *apuntactual* contendrá la dirección del registro de creación más reciente. Las siguientes proposiciones crean dinámicamente el primer registro y asignan a los apuntadores los valores iniciales que corresponden al inicio de la lista:

```

new (apuntinic);
apuntactual := apuntinic
  
```

Ahora, para crear una lista encadenada con tres registros más y el valor NIL en el apuntador del último registro, se ejecutará el siguiente código:

```

FOR índice := 1 TO 3 DO
BEGIN
    new (apuntactual ↑ .sigreg);      (* obtener registro nuevo *)
    apuntactual := apuntactual ↑ .sigreg (* avanzar el apuntador *)
END;
apuntactual ↑ .sigreg := NIL        (* terminar la lista *)
  
```

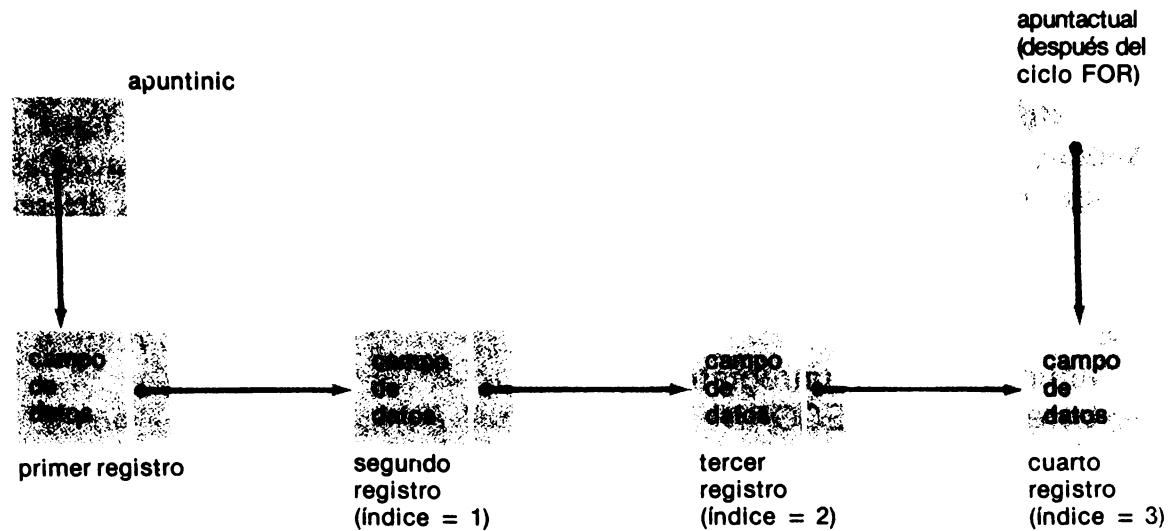
La figura 13-2 muestra la construcción de esta lista encadenada.

La proposición del ciclo FOR

```

apuntactual := apuntactual ↑ .sigreg
  
```

avanzará el apuntador cada vez que se ejecute el ciclo. La lista encadenada se termina con el valor NIL. Tómese en cuenta que el ciclo anterior simplemente crea la lista encadenada al asignar memoria para cada registro y avanzar el apuntador al siguiente registro. Se podía haber incluido una proposición *read* para introducir los datos a cada registro.



**Figura 13-2** Lista encadenada.

### Búsquedas en listas encadenadas

Supóngase que en el ejemplo anterior se creó una lista encadenada en la que cada registro contiene un dato de caracteres y un apuntador al siguiente registro. En ese caso la definición de tipo sería la siguiente:

```
TYPE
    apuntador = ↑ unregistro;
    unregistro = RECORD
        datos: char;
        sigreg: apuntador
    END;
```

Supóngase además que se asignó al apuntador *apuntinic* la dirección del comienzo de la lista encadenada y que el campo *sigreg* del último registro tiene el valor NIL. Considérese que existe un número no conocido de registros y que la lista encadenada no está vacía, esto es, existe por lo menos un registro. Consultese la figura 13-2, donde se sustituyen los campos de datos por un dato de carácter. Considérese el siguiente problema de búsqueda en la lista encadenada.

### Problema 13.3

*Escríbase un procedimiento en Pascal llamado buscar que exhiba un mensaje que indique si un valor de carácter llamado código se encuentra en uno de los registros de la lista encadenada*

La solución se muestra en el siguiente procedimiento, el cual supone la definición de tipo que ya se describió:

```
PROCEDURE buscar (código: char;
    apuntinic: apuntador;
    VAR apuntactual: apuntador);
```

(\* Buscar código en la lista encadenada \*)  
(\* y exhibir un mensaje apropiado \*)

BEGIN

```

apuntactual := apuntinic; (* valor inicial apuntador actual *)
WHILE apuntactual^.datos <> código AND (* no se halló código *)
    (apuntuactual^.sigreg <> NIL) DO (* y no termina la lista *)
        apuntactual := apuntactual^.sigreg; (* avanzar *)
IF apuntactual^.datos = código (* ¿se halló el código? *)
THEN writeln ('El código está en la lista.') (* sí, éxito *)
ELSE writeln ('El código r.o está en la lista.') (* no, fracaso *)

```

END;

## Inserción en listas encadenadas

Supóngase que se tiene un arreglo clasificado de enteros y es necesario insertar un nuevo entero en la posición correcta del arreglo. Como ya se mencionó, este proceso puede consumir mucho tiempo, sol e todo cuado es preciso desplazar o copiar muchos elementos. En cambio, si la estructura de datos es una lista encadenada, la inserción se puede hacer mediante el ajuste de apuntadores. Por ejemplo, considérese la lista encadenada que se muestra en la figura 13-3a, donde los valores enteros de los registros están en orden ascendente como sigue: 2, 6, 12, 15. Cada uno de los apuntadores de los registros apunta al siguiente registro, a excepción, naturalmente, del último registro que contiene al apuntador NIL.

Supóngase que se debe insertar el valor entero 8. Como puede verse en la figura 13-3b, se crea un registro nuevo que contenga el valor 8. El apuntador del registro que contiene el valor 6 apunta ahora al registro nuevo y el apuntador del registro nuevo apunta al registro que contiene el valor 12. No es preciso desplazar componentes como en el caso de los arreglos. Si el lector medita brevemente sobre este ejemplo, se convencerá de la eficiencia de este método para insertar en una lista encadenada, en comparación con la inserción en un arreglo.

Considérese ahora que se tiene una lista encadenada con las declaraciones anteriores y en la que los valores de los registros están en orden ascendente. Supóngase que se debe insertar un registro que contiene un valor entero llamado *nuevonúm* en la posición correcta de la lista encadenada de manera que los valores de los registros sigan en orden. Se supondrá que se tienen dos apuntadores llamados *menor* y *mayor* que apuntan a los registros que preceden y siguen inmediatamente a *nuevonúm*. En el ejemplo, *menor* apuntaría al registro que contiene el valor 6 y *mayor* apuntaría al registro que contiene 12. Ya se pueden escribir las proposiciones en Pascal que realizan la inserción.

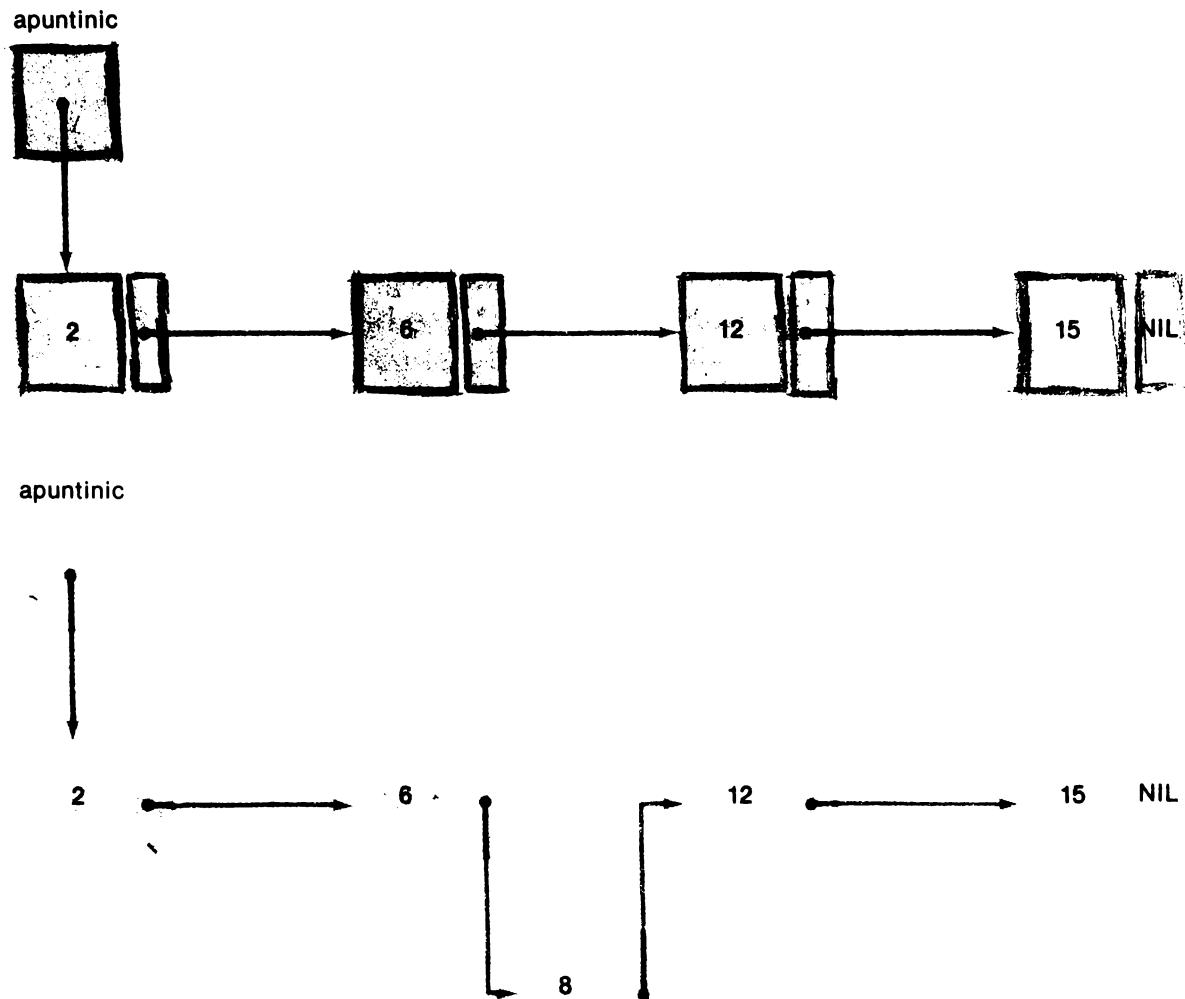
El primer paso es asignar dinámicamente memoria para un registro nuevo y almacenar el valor *nuevonúm* en el registro mediante la variable de apuntador *apuntinsert* (supóngase que se declaró *apuntinsert* como de tipo apuntador):

```

new (apuntinser);
apuntinser^.datos := nuevonúm
apuntinser

```

Ahora ya se pueden ajustar los apuntadores mediante las variables de apuntador *menor* y *mayor*:



**Figura 13-3** Inserción en listas encadenadas

```
menor ↑ .sigreg := apuntinsert;
apuntinsert ↑ .sigreg := mayor
```

Cabe subrayar que en este ejemplo se supuso que *nuevonúm* queda entre otros dos valores en la lista encadenada. Puede suceder que *nuevonúm* sea menor que cualquiera de los valores de la lista o, quizás, mayor que cualquiera de ellos. Al escribir un programa completo de inserción en listas encadenadas sería preciso tomar en cuenta estos casos.

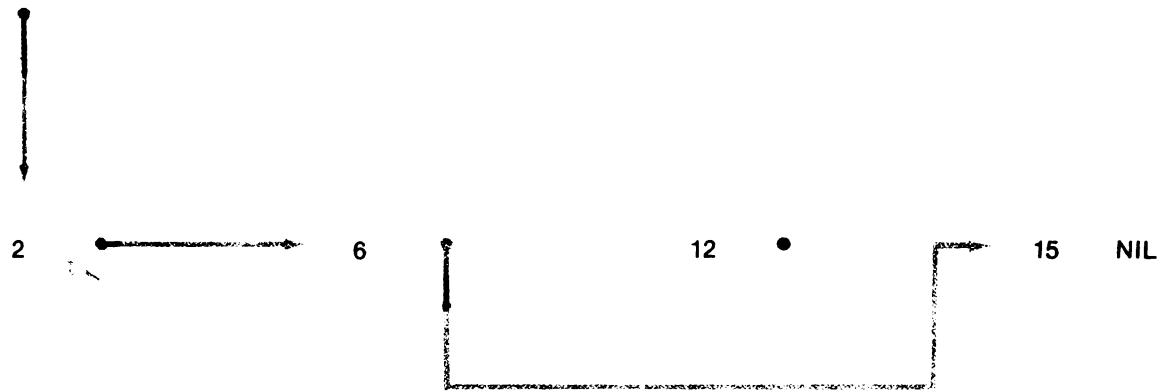
### Eliminación en listas encadenadas

Se continuará con el mismo ejemplo para ver lo fácil que es eliminar un elemento de la lista. La figura 13-4a muestra la lista encadenada original con los valores 2, 6, 12 y 15. Supóngase que se va a eliminar el valor 12. La figura 13-4b muestra los ajustes a los apuntadores.

El apuntador del registro que contiene el valor 6 se ajusta de manera que apunte al registro que contiene el valor 15, y el registro que contiene el valor 12 se libe-



apuntinic



**Figura 13-4** Eliminación en listas encadenadas

ra mediante el procedimiento *dispose*. Supóngase que la variable de apuntador *apuntactual* apunta al registro que se va a eliminar y *menor* es una variable de apuntador que señala al registro que precede inmediatamente al registro que se va a quitar. En ese caso, la eliminación se realiza con las siguientes proposiciones:

*menor*  $\uparrow$  .sigreg := *apuntactual*  $\uparrow$  .sigreg;  
*dispose* (*apuntactual*)

Una vez más, esto no funcionará si el registro que se ha de eliminar es el primero, ya que no existe un registro que le preceda inmediatamente. Este caso se debe manejar por separado en un programa completo que haga eliminaciones en una lista encadenada (véase el ejercicio 5 de la sección 13.3). Finalmente, si se va a eliminar el último registro, el apuntador del penúltimo registro deberá recibir el valor NIL.

### EJERCICIOS DE LA SECCIÓN 13.3

- 1 ¿Por qué se usó el valor NIL para marcar el final de la lista encadenada?
- 2 Escríbase un segmento en Pascal similar al que se presentó al principio de la sección para crear una lista encadenada que incluya una tercera variable de apuntador que apunte al último registro de la lista encadenada.

- 3 Escribáse un procedimiento similar al que se escribió para el problema 13.3 que exhiba el contenido de los valores contenidos en cada uno de los registros de la lista encadenada.
- 4 Consúltese el material sobre inserción en listas encadenadas para escribir un segmento en Pascal que inserte un registro nuevo al principio de la lista encadenada.
- 5 Consúltese el material sobre eliminación en listas encadenadas para escribir un segmento en Pascal que elimine el primer registro de una lista encadenada.

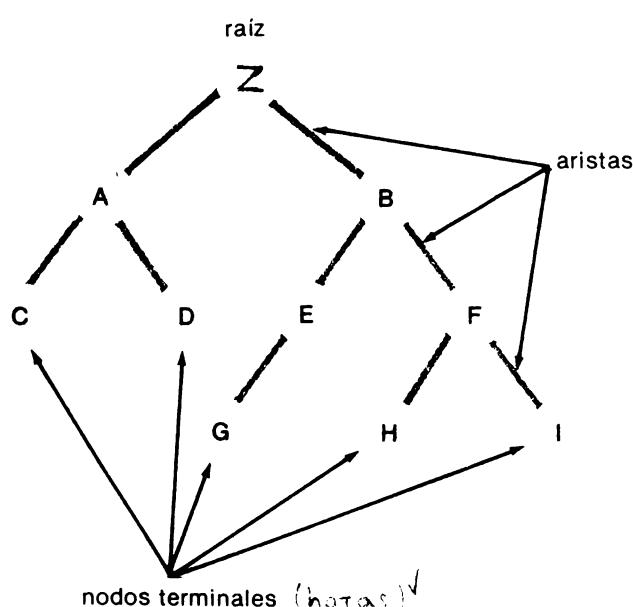
## SECCIÓN 13.4 ÁRBOLES

Si se examinan las figuras de este texto que contienen diagramas de diseño descendente, se observará que tienen una *estructura de árbol*, es decir, existe un solo cuadro en el nivel superior del diagrama que se llama *raíz* del árbol. La raíz del diagrama de árbol está conectada a otros cuadros en el siguiente nivel abajo de la raíz. Cada uno de estos cuadros puede, a su vez, estar conectado a otros cuadros en el nivel inmediato inferior. La figura 13-5 muestra un ejemplo de estructura general de árbol en la que se sustituyeron los cuadros por círculos con etiquetas.

Los círculos con etiquetas se llaman *nodos* y se conectan mediante aristas. La raíz de la figura 13-5 tiene la etiqueta Z y tiene dos hijos, un hijo izquierdo llamado A y un hijo derecho llamado B. El nodo A también tiene dos hijos, C y D. Sin embargo, C y D no tienen hijos y se denominan *nodos terminales* u *hojas*.

Si un árbol tiene la propiedad de que ninguno de los nodos tiene más de dos hijos, se llama *árbol binario*. La figura 13-5 muestra un árbol binario, ya que todos los nodos tienen cero, uno o dos hijos. Los árboles binarios son estructuras de datos que permiten usar técnicas muy eficientes de almacenamiento y recuperación de datos, por lo que son de suma importancia en computación.

**Figura 13-5** Árbol binario.



Al construir un árbol binario en Pascal, se emplean registros para representar a los nodos, y apuntadores para representar a las aristas. En los árboles binarios se necesitan dos apuntadores (izquierdo y derecho) para apuntar a los hijos, si existen. Se asignará el valor NIL a un apuntador si no hay un hijo en la dirección correspondiente. La definición del nodo en un árbol binario sería

#### TYPE

```

apuntador = ↑ nodo;
nodo = RECORD
    campo: tipo;           (* campo(s) de datos *)
    izq: apuntador;       (* apunta al hijo izquierdo *)
    der: apuntador;       (* apunta al hijo derecho *)
END;
  
```

La diferencia principal entre la lista encadenada que se mostró antes y el árbol binario que se desarrolló aquí es que el árbol binario utiliza dos apuntadores.

#### Búsqueda en árboles binarios

Ahora se verá por qué son tan útiles los árboles binarios. Súpongase que se tiene un arreglo que contiene los siguientes caracteres

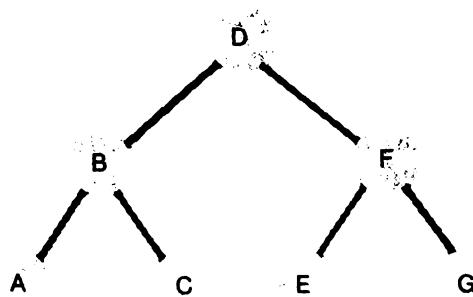
D    F    E    B    A    C    G

Recuérdese que, en el capítulo 9 (“Arreglos”), para determinar si un carácter está en la lista se puede hacer una búsqueda lineal (ya que los caracteres no están en orden alfabético). Sin embargo, si el arreglo es muy grande, las búsquedas lineales son poco eficientes, sobre todo si el elemento no está en la lista. Para determinar si el elemento no está en la lista sería menester examinar todo el arreglo. Es posible clasificar el arreglo y emplear una búsqueda binaria, lo cual es muy eficiente. Pero supóngase que se insertan y eliminan datos continuamente. En ese caso se tendrá el problema de inserción y eliminación de elementos de un arreglo que se analizó en la sección anterior.

Lo que se necesita es una estructura de datos en la que los elementos se puedan localizar, eliminar e insertar de manera eficiente. Una solución a este problema es una estructura dinámica de datos conocida como **árbol de búsqueda binaria**. Primero es preciso construir la estructura de datos, para lo cual se emplea el siguiente algoritmo:

PASO 1 Se crea la raíz del árbol de búsqueda binaria con el primer dato.

PASO 2 Cada uno de los datos subsecuentes se compara con la raíz. Si el elemento precede alfabéticamente al valor de la raíz, entonces el dato se convertirá en su hijo izquierdo o bien se comparará con el hijo izquierdo ya existente, y se repetirá el proceso. Si el dato sigue alfabéticamente a la raíz, o bien se convertirá en un hijo derecho o se comparará con el hijo derecho ya existente, y el proceso se repetirá.



**Figura 13-6** Árbol de búsqueda binaria.

Ahora se volverá al ejemplo y se aplicará el algoritmo. El carácter D se usará como valor de la raíz del árbol. El siguiente elemento, F, sigue a D, por lo que se convierte en hijo derecho de la raíz. En seguida se compara E con la raíz. Puesto que E sigue a D, se compara con el hijo derecho, F. Dado que E precede a F, se convierte en hijo izquierdo de F. El proceso se repite con cada uno de los datos restantes. La figura 13-6 muestra el árbol de búsqueda binaria resultante.

Ahora ya se sabe cómo crear un árbol de búsqueda binaria a partir de un conjunto de datos. La adición de datos nuevos al árbol se logra mediante la ejecución del paso 2 del algoritmo usado inicialmente para construir el árbol. No se hablará aquí de los algoritmos para eliminar elementos de un árbol de búsqueda binario.

Es importante destacar que el hijo izquierdo de una raíz de todos sus descendientes (tanto derechos como izquierdos) tienen datos menores que el de la raíz. El hijo derecho y todos sus descendientes tienen datos mayores que el de la raíz.

El algoritmo para localizar un dato, llamado *llave*, en un árbol de búsqueda binaria comienza por comparar *llave* con la raíz. Si *llave* es igual a la raíz, la búsqueda termina con éxito. Si *llave* precede a la raíz, la búsqueda continúa con el hijo izquierdo. En caso contrario, la búsqueda continúa con el hijo derecho, ya que *llave* seguirá a la raíz. Si no quedan hijos por comparar, *llave* no estará en el árbol de búsqueda. El siguiente problema requiere el uso de este algoritmo.

#### Problema 13.4

*Escríbase una función en Pascal llamada búsbin que determine si una variable de carácter llamada llave se encuentra en un árbol de búsqueda binaria. El valor de búsbin deberá ser NIL si el elemento no se encontró, o un apuntador al nodo localizado. Supóngase que los nodos del árbol se definieron así:*

```

TYPE
  apuntador = ^nodo;
  nodo = RECORD
    letra: char;
    izq: apuntador;
    der: apuntador
  END;
  
```

Ésta es la solución del problema:

**FUNCTION** búsbín (árbolbin: apuntador; llave: char): apuntador;  
 (\* Buscar llave en árbolbin. Si se encuentra, el resultado es el \*)  
 (\* apuntador al registro correspondiente; si no, el resultado es NIL \*)

VAR listo: boolean;

BEGIN

```

    listo := false;
    REPEAT
        IF árbolbin = NIL
        THEN listo := true
        ELSE IF llave < árbolbin↑ .letra
            THEN árbolbin := árbolbin↑ .izq (* llave < valor *)
            ELSE IF llave > árbolbin↑ .letra (* llave > valor *)
                THEN árbolbin := árbolbin↑ .der
            ELSE listo := true (* llave = valor *)
    UNTIL listo;
    búsbín := árbolbin
END;
```

## Recorridos

En muchos problemas es importante *recorrer* un árbol binario, esto es, puede ser necesario examinar todos los nodos del árbol, por ejemplo para hacer una lista de los valores de los nodos o efectuar alguna otra operación sobre los datos.

Existen muchas formas de visitar los nodos de un árbol binario. Por ejemplo, existen seis formas diferentes de recorrer los nodos de un árbol binario que tiene solamente tres nodos. La figura 13-7 muestra un árbol binario con tres nodos marcados con A, C y E. También se indican los órdenes en que se pueden recorrer los nodos. Por ejemplo, el orden ACE significa visitar la raíz A, visitar después C y por último E.

En la mayor parte de los casos, empero, sólo se emplean tres órdenes de recorrido, y se ponen en práctica fácilmente si se visita la raíz y se recorren el *subárbol derecho* y el *subárbol izquierdo* (aunque no necesariamente en ese orden). El subárbol izquierdo de la raíz es su hijo izquierdo y todos sus descendientes. El subárbol derecho de la raíz se define de manera similar. Esta definición se aplica también a todos los demás nodos. Todos los subárboles, con excepción del subárbol vacío (NIL), tienen también una raíz.

Los tres recorridos de árboles binarios de uso más frecuente recorren los subárboles izquierdos antes que los subárboles derechos. Éstos son los recorridos *enor-*

**Figura 13-7** Recorridos de árboles.



Seis recorridos posibles: ACE, AEC, CAE, CEA, EAC, ECA

*den, postorden* y *preorden*. Los nombres indican el punto en que se visita la raíz durante el recorrido. El enorden visita la raíz a mitad del recorrido, preorden visita la raíz al principio y postorden visita la raíz al último.

El recorrido enorden tiene el siguiente algoritmo recursivo:

PASO 1 Recorrer el subárbol izquierdo.

PASO 2 Visitar la raíz.

PASO 3 Recorrer el subárbol derecho.

Al aplicar este algoritmo al árbol de búsqueda binaria de la figura 13-6, se visitarán los nodos en el orden

A      B      C      D      E      F      G

Obsérvese que este algoritmo de recorrido visita los nodos de un árbol de búsqueda binaria en orden ascendente. Esta característica hace que el recorrido enorden sea el tipo de recorrido que se usa con mayor frecuencia. El siguiente procedimiento recursivo en Pascal lleva a la práctica este algoritmo. El procedimiento *visitar* sirve para realizar cualquier operación que se desee en el nodo al que hace referencia su parámetro, y no se muestra aquí. Supóngase que se hacen las mismas declaraciones que en el caso del problema 13.4.

```
PROCEDURE recorrido (árbolbin: apuntador);
(* Hacer el recorrido enorden de un árbol binario. *)
BEGIN
```

```
    IF árbolbin <> NIL
        THEN BEGIN
            recorrido (árbolbin↑ .izq); (* subárbol izquierdo *)
            visitar (árbolbin); (* visitar raíz del subárbol *)
            recorrido (árbolbin↑ .der) (* subárbol derecho *)
        END
    END;
```

El recorrido en postorden (o ascendente) tiene el siguiente algoritmo recursivo:

PASO 1 Recorrer el subárbol izquierdo.

PASO 2 Recorrer el subárbol derecho.

PASO 3 Visitar la raíz.

Cuando se aplica al árbol de búsqueda binaria de la figura 13-6, los nodos se visitan en este orden:

A      C      B      E      G      F      D

Por último, el recorrido en preorden (o de adentro hacia afuera) tiene el siguiente algoritmo recursivo:

PASO 1 Visitar la raíz.

PASO 2 Recorrer el subárbol izquierdo.

PASO 3 Recorrer el subárbol derecho.

D    B    A    C    F    E    G

## EJERCICIOS DE LA SECCIÓN 13.4

- 1 Constrúyase un árbol de búsqueda binaria para las siguientes letras siguiendo el orden que se da:

H    J    A    C    B    Z    T

- 2 Constrúyase un árbol de búsqueda binaria para la siguiente secuencia de enteros:

7    3    6    2    1    4    8    5

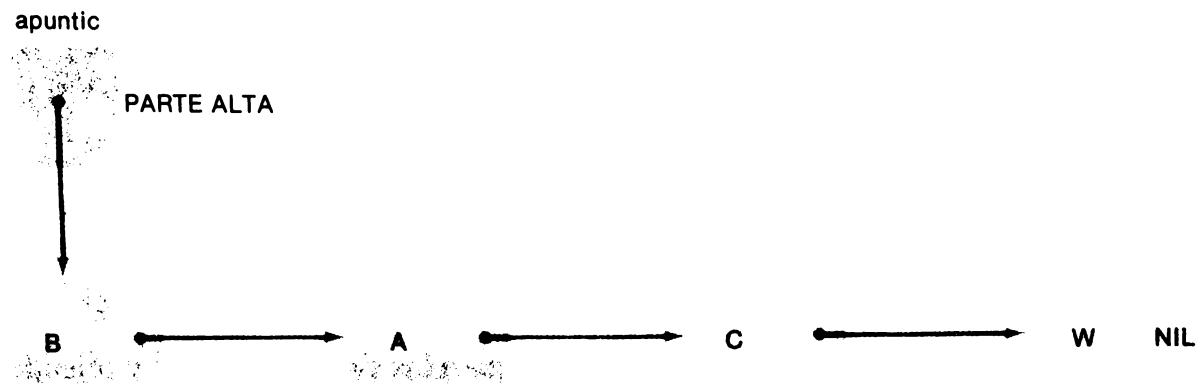
- 3 Aplíquense los tres algoritmos de recorrido a los árboles de búsqueda binaria producidos en los ejercicios 1 y 2.
- 4 Escríbase un procedimiento para insertar un nuevo carácter llamado *nuevocar* en el árbol de búsqueda binaria utilizado en la función *buscar*.
- 5 Escríbase un procedimiento recursivo similar al procedimiento *recorrido* que visite los nodos de un árbol binario en recorrido postorden.
- 6 Escríbase un procedimiento recursivo similar al procedimiento *recorrido* que visite los nodos de un árbol binario en recorrido preorden.

## SECCIÓN 13.5 PILAS Y COLAS

En esta sección se presenta un análisis breve de dos estructuras de datos dinámicas, las pilas y las colas, que se emplean en compiladores, sistemas operativos y otras aplicaciones. Una *pila* es una estructura de datos dinámica que tiene una “parte superior” con la propiedad de que el último elemento que se “introduce” (*push*) a la pila es el primer elemento que se “extrae” (*pop*) de la pila. El acceso a los elementos se realiza por un extremo, que es la *parte superior* de la pila. Piénsese en una pila de platos en una cafetería. El plato de arriba es el único que está accesible, pero tan pronto como se quita, el plato que está inmediatamente debajo sube a la parte superior.

Las pilas se pueden representar mediante listas encadenadas en las que el apuntador inicial funciona como parte superior de la pila y el último apuntador es igual a NIL, como siempre. La figura 13-8 muestra una pila creada mediante la inserción de los siguientes caracteres en el orden que se muestra:

W    C    A    B



**Figura 13-8 Pila.**

Obsérvese que el último elemento que se inserta es B, que está en la parte superior de la pila. Si se hace la siguiente definición de tipo

```
TYPE
  apunpila = ^regpila;
  regpila = RECORD
    símbolo: char;
    sigreg: apunpila
  END;
```

entonces el siguiente procedimiento introducirá un nuevo carácter llamado *nuevaleta* en la parte superior de la pila.

```
PROCEDURE introduce (nuevaleta: char; VAR alta: apunpila);
(* Introducir nuevaleta en la parte superior de la pila. *)
VAR
  apuntemp: apunpila;
BEGIN
  new (apuntemp);          (* crear nuevo registro para parte alta *)
  apuntemp^.símbolo := nuevaleta; (* guardar nuevaleta *)
  apuntemp^.sigreg := alta;   (* apuntar a parte alta anterior *)
  alta := apuntemp          (* actualizar apuntador de la parte alta *)
END;
```

El siguiente procedimiento llamado *saca* eliminará el elemento *salcar* de la parte superior de la pila y ajustará el apuntador de la parte alta en la forma apropiada. Puesto que no se puede eliminar un elemento de una pila vacía, se exhibirá un mensaje de error si la pila está vacía en ese momento.

```
PROCEDURE saca (VAR salcar: char; VAR alta: apunpila);
(* Sacar de la pila el elemento de la parte superior *)
VAR
  apuntemp : apunpila;
```

```

BEGIN
  IF alta = NIL
  THEN writeln ('Error: la pila está vacía.')
  ELSE BEGIN
    salcar := alta↑ .símbolo; (* obtener último carácter *)
    apuntemp := alta; (* guardar apuntador a parte alta *)
    alta↓ := alta↑ .sigreg; (* ajustar apuntador a parte alta *)
    dispose (apuntemp)      (* eliminar registro antiguo *)
  END
END;
  
```

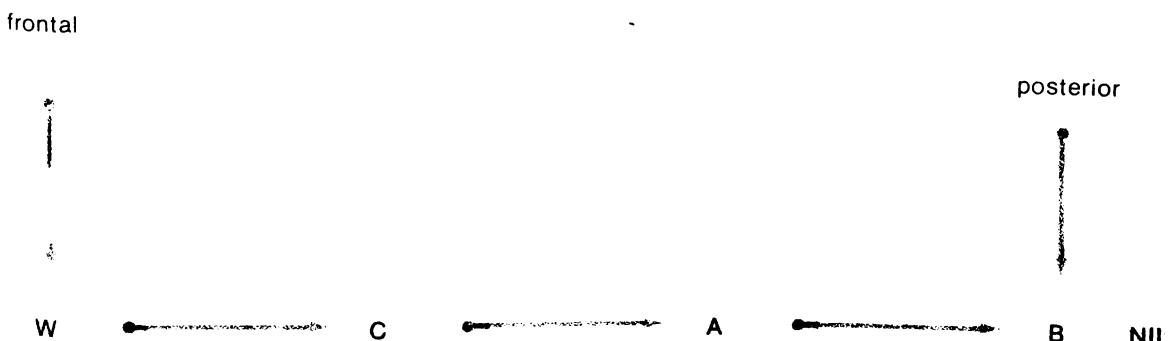
Las pilas tienen muchas aplicaciones en computación. Por ejemplo, al ejecutar procedimientos o funciones recursivos, la computadora emplea una pila para seguir la pista de las invocaciones recursivas. Además, cuando la computadora evalúa expresiones, las traduce en una forma equivalente pero sin paréntesis. Una vez hecho esto, la computadora puede evaluar las expresiones de manera eficiente al emplear una pila como almacenamiento temporal.

Las *colas* son estructuras de datos en las que los elementos se introducen por un extremo y se extraen por el otro. El primer elemento que se inserta es siempre el primer elemento que se saca, a diferencia de las pilas, en las que el primer elemento que se inserta es siempre el último elemento que se saca. Esto se parece a las filas, o colas, que se forman en las cajas registradoras de un supermercado. La primera persona que se forma es normalmente la primera que sale. La figura 13-9 muestra una cola que se crea mediante la inserción de los siguientes caracteres en el orden que se indica (adviértase la necesidad de usar dos apuntadores, uno frontal y uno posterior).

W      C      A      B

Obsérvese que el elemento W que se insertó primero está al frente de la cola y el elemento B que se insertó al último está hasta atrás. El procedimiento *encola*, que se muestra a continuación, puede servir para agregar un elemento al final de la cola. Cuando la cola está vacía (es decir, *frontal* = NIL y *posterior* = NIL) es preciso asignar a ambos apuntadores la dirección de un elemento recién asignado. Si la cola no está vacía, se agregará un elemento nuevo después de *posterior* ↑,

**Figura 13-9** Cola.



y *posterior* se ajustará de manera que apunte al elemento que se acaba de agregar. Supóngase que se declararon dos apuntadores, *frontal* y *posterior*, de tipo *apuncola* (que se define como se indica en seguida).

## TYPE

```

apuncola = ↑ regcola;
regcola = RECORD
    símbolo: char;
    sigreg: apuncola
END;

PROCEDURE encola (nuevaleta: char; VAR frontal, posterior:apuncola);
(* Agregar nuevo elemento al final de la cola *)
VAR
    coltemp: apuncola;
BEGIN
    new(coltemp); (* apartar registro nuevo *)
    coltemp↑ .símbolo := nuevaleta; (* guardar letra nueva *)
    IF posterior<>NIL (* agregar al final de la *)
        THEN posterior↑ .sigreg := coltemp; (* cola existente *)
        coltemp↑ .sigreg := NIL; (* terminar la cola *)
        posterior := coltemp; (* ajustar apuntador posterior *)
        IF frontal = NIL (* si antes estaba vacía *)
            THEN frontal := posterior (* ajustar apuntador frontal *)
    END;

```

Para sacar un elemento del frente de la cola se usa un procedimiento similar a *saca*, que sirve para sacar un elemento de una pila. No obstante, en el procedimiento *sacola*, que se muestra en seguida, es preciso tener en cuenta el caso de que la cola quede vacía, pues será necesario asignar el valor NIL al apuntador posterior.

```

PROCEDURE sacola (VAR salcar: char; VAR frontal, posterior: apuncola);
(* Sacar un elemento del frente de la cola. *)
VAR
    coltemp: apuncola;
BEGIN
    IF frontal = NIL
        THEN writeln ('Error. La cola está vacía.')
    ELSE BEGIN
        salcar := frontal↑ .símbolo; (* guardar elemento *)
        coltemp := frontal (* guardar apuntador frontal *)
        frontal := frontal↑ .sigreg (* ajustar frontal *)
        dispose (coltemp); (* eliminar registro antiguo *)
        IF frontal := NIL (* si la cola queda vacía *)
            THEN posterior := NIL (* ajustar apuntador posterior *)
    END

```

- 1 Sin emplear apuntadores, describase la forma de crear una pila mediante un arreglo. ¿Cómo se sabría dónde está la parte superior de la pila?
- 2 Escríbase una función de resultado booleano que valga *true* si una pila está vacía y *false* en caso contrario. Supóngase que se hicieron las declaraciones de los procedimientos *introduce* y *saca*.
- 3 ¿Es necesario verificar si una pila está llena o no? Explíquese la respuesta.
- 4 Escríbase un procedimiento para exhibir todos los elementos de una cola de adelante hacia atrás. Supóngase que se hicieron las declaraciones de los procedimientos de manejo de colas de esta sección.
- 5 Escríbase un procedimiento que vacíe una cola. Si la cola no estaba ya vacía, es preciso asegurarse de que todas las localidades de memoria usadas queden libres.

## SECCIÓN 13.6 ABSTRACCIÓN DE LOS DATOS

En este libro se ha hecho hincapié en la estrategia descendente del diseño de algoritmos. En ella, muchos detalles de la solución del problema se postergan hasta que sea necesario llevarlos a la práctica dentro del proceso de refinación por pasos. El lector ya sabe que los programas manipulan datos que se emplean para modelar objetos, por lo que requieren el diseño de las estructuras de datos que se emplean en el modelo. En este capítulo se han estudiado varios tipos de estructuras de datos. La pregunta inmediata es: ¿cómo se diseña y se pone en práctica la estructura de datos apropiada para un modelo determinado? Una respuesta a esta pregunta utiliza la técnica conocida como *abstracción de los datos*.

La estrategia descendente también se puede aplicar al diseño de estructuras de datos. Esto es lo que se llama abstracción de los datos. La estrategia parte de una estructura de datos general o abstracta, y no se especifican los detalles prácticos. Basta con especificar las funciones que el programa utilizará para manipular la estructura de datos. En el siguiente refinamiento se proporcionarán más detalles de la estructura de datos, como el seudocódigo de las funciones. En el refinamiento final se especificarán los detalles prácticos de las funciones y la organización de memoria para la estructura de datos.

Por ejemplo, considérese el problema de ordenar alfabéticamente una lista de nombres. Ya se han descrito varios algoritmos de clasificación, cualquiera de los cuales puede servir para resolver este problema. Ahora se estudiará el diseño de la estructura de datos. En el nivel más alto de abstracción de los datos, se tiene una lista de nombres, todos, con excepción del último, seguidos de un solo nombre y todos, con excepción del primero, precedidos de un solo nombre. En el siguiente nivel se requieren más detalles para especificar la estructura de los datos. Se sabe que es menester poder comparar los nombres individuales en la estructura de datos (para la clasificación), y también es necesario poder cambiar la posición de un nombre en la estructura de datos en relación con su predecesor y sucesor en la lista. Éstas son las únicas funciones que se requieren para ordenar la lista de nombres. Téngase en mente que en este nivel no se han especificado en absoluto los detalles del almacenamiento real de los nombres. En los niveles más detallados de

la especificación será necesario decidir si conviene usar un arreglo de registros, en el que cada registro contiene un nombre (representado mediante un arreglo de caracteres), o quizá un arreglo de arreglos de cadenas de caracteres (es decir, un arreglo bidimensional de caracteres). Por último, es preciso llevar a la práctica las funciones y el almacenamiento de la lista mediante la escritura de código en Pascal para cada una de las funciones que se identificaron, así como las declaraciones de la estructura de datos. Aquí se proporciona un resumen de estas observaciones.

<i>nivel</i>	<i>abstracción de los datos</i>	<i>ejemplo</i>
Más alto	Estructura de datos generalizada Funciones requeridas para manipular los datos	Una lista de nombres Comparar dos nombres, mover un nombre.
Más bajo	Procedimientos, funciones y declaraciones en Pascal	VAR nombre: ARRAY [1..100] OF PACKED ARRAY [1..25] OF char; FUNCTION compnom (...) FUNCTION muevenom (...)

La estructura de datos específica que se deba usar dependerá de muchos factores, como son las limitaciones de tiempo y memoria, las especificaciones y restricciones del programa, etc. Lo que es importante recordar es que, si se aplica efectivamente la abstracción de los datos, es posible simplificar el proceso de resolución del problema en forma similar a lo que sucede al elaborar algoritmos en forma descendente.

## SECCIÓN 13.7 TÉCNICAS DE PRUEBA Y DEPURACIÓN

Los apuntadores son una fuente potencial de errores con motivo de las reglas que gobiernan su uso. Por ejemplo, para poder hacer referencia a una dirección de memoria, no basta con declarar una variable de apuntador. Es necesario asignarle el valor de un apuntador que ya señale a una dirección de memoria, o bien usarlo como argumento del procedimiento *new*. La variable de apuntador contiene una dirección de memoria. Para tener acceso a la variable de referencia que está en esa dirección de memoria, se debe agregar el símbolo especial  $\uparrow$  a la variable de apuntador. Si el apuntador no está definido o es NIL, al tratar de tener acceso a la variable de referencia se provocará un error de ejecución.

Si se invoca el procedimiento *dispose* para un apuntador, éste no estará definido lógicamente, y no se le deberá usar para tener acceso a una variable de referencia. Por desgracia, algunas versiones de Pascal permiten usar la variable de apuntador para tener acceso a la misma variable de referencia, aun después de aplicarse el procedimiento *dispose*, lo que puede llevar a usar memoria que ya se había apartado para otro uso, quizás para una variable dinámica de un tipo diferente.

En ocasiones se pueden presentar errores al invocar el procedimiento *new* en un ciclo infinito, o al intentar apartar más memoria de la que se dispone en el sistema de cómputo. La memoria también puede agotarse si no se emplea el procedimiento *dispose* para liberar la memoria que no se usa. No hay forma de que el sistema

de Pascal sepia que ya no se necesitan ciertas variables dinámicas, a menos que esto se especifique explícitamente mediante el uso de *dispose*.

Normalmente no se recomienda pasar estructuras de datos grandes como parámetros de valor ya que requieren el copiado de las estructuras completas. En el caso de las estructuras de datos dinámicas, esta situación no es grave, ya que basta con pasar los apuntadores de la estructura de datos. Los apuntadores precisan muy poca memoria y copiarlos representa una operación muy eficiente. No obstante, el programador debe tener cuidado al pasar como parámetros las estructuras de datos dinámicas, ya que los cambios que se efectúan en los elementos de la estructura de datos afectan a la estructura original, no a una copia.

Dado que las estructuras dinámicas, como son las colas y las listas encadenadas, pueden estar vacías, es importante incluir las pruebas necesarias para verificar si éste es el caso. Si no se hace esto, es posible que se produzcan errores de ejecución o resultados incorrectos.

Si se presentan problemas en programas que usan estructuras de datos dinámicas, a menudo resulta conveniente exhibir los datos almacenados en la estructura y la relación entre los elementos. Por ejemplo, si se usa un árbol de búsqueda binaria, es posible emplear cualquiera de los tres procedimientos de recorrido para visitar todos los nodos del árbol, y el procedimiento *visitar* incluiría únicamente una proposición *write* o *writeln* para exhibir el campo (o campos) de datos del nodo.

En seguida se proporcionan algunos recordatorios importantes que pueden ser útiles al probar y depurar programas.

## RECORDATORIOS DE PASCAL

- Encerrar los constructores de conjuntos en paréntesis cuadrados:  
['A', 'E', 'I', 'O', 'U']
- No es posible tener acceso directo a los elementos individuales de un conjunto. Úsese el operador IN para determinar membresía.
- Para manipular un conjunto es preciso especificar sus elementos iniciales.
- Los operadores de conjuntos, “+” “-” y “\*”, cuando se aplican a conjuntos, no son iguales a los operadores aritméticos correspondientes, pero tienen la misma prioridad.
- Las variables de apuntador contienen direcciones de memoria.
- Las variables de apuntador no se pueden leer de archivos de texto ni grabarse en ellos.
- Las variables de apuntador del mismo tipo se pueden comparar para determinar igualdad o desigualdad, o asignarse entre sí.
- Para tener acceso a la variable almacenada en la dirección de memoria que indica una variable de apuntador, se debe usar el nombre de la variable de apuntador seguida de “↑”.
- Al definir los tipos de datos de apuntador no debe olvidarse la inclusión del símbolo “↑” antes del identificador de tipo:  
TYPE apuntador = ↑ integer;
- No conviene tener variables de apuntador no definidas; utilícese el valor NIL en caso

- El uso de *dispose (apun)* lógicamente dejará a *apun* sin definir; una referencia a *apun* ↑ provocará un error lógico o un error de ejecución.

## SECCIÓN 13.8 REPASO DEL CAPÍTULO

En este capítulo se analizó el tipo de datos estructurado de conjunto. También se habló de las operaciones de conjuntos unión, intersección y diferencia. Se vio la aplicación de los operadores relacionales estándar a los conjuntos.

Pascal incluye un tipo de datos predefinido conocido como apuntador. Mediante el procedimiento estándar *new* es posible apartar dinámicamente una variable a la cual se puede hacer referencia con el nombre de la variable de apuntador seguido del símbolo “↑”. Se explicó la forma de usar los apuntadores para construir estructuras de datos dinámicas, como son listas encadenadas, árboles binarios, pilas y colas. Se escribieron procedimientos para buscar, insertar y eliminar elementos y recorrer árboles binarios.

Ahora se brinda un resumen de las características de Pascal que se analizaron en el capítulo; puede ser útil como referencia en el futuro.

## REFERENCIAS DE PASCAL

- 1 Conjunto. Tipo de datos estructurado que consta de cierto número de elementos distintos elegidos entre los del tipo base ordinal. Ejemplo:

```
TYPE conjnúm = SET OF 1..128;
VAR números: conjnúm;
```

- 1.1 Operadores de conjuntos cuyo resultado es un conjunto:

- + Unión
- Diferencia
- \* Intersección

- 1.2 Operadores relacionales con resultados booleanos:

- = Igualdad de conjuntos
- <> Desigualdad de conjuntos
- <=, >= Subconjuntos o igual
- IN Membresía

- 2 Apuntador. Tipo de datos que contiene la dirección de la variable de referencia.

### TYPE

```
apuntador = ↑ nodo;
nodo = RECORD
    código: char;
    izq: apuntador;
    der: apuntador
END;
```

- 2.1 Si *apun* es el nombre de una variable de apuntador, *apun* ↑ será el nombre de la variable de referencia.
- 2.2 Procedimientos estándar:  
*New (apun)*. Aparta dinámicamente una localidad de memoria y asigna su dirección a *apun*.  
*Dispose (apun)*. Indica que la dirección memoria a la que apunta *apun* queda libre, lo que hace que *apun* no está definido lógicamente.

**3 Algunas estructuras de datos dinámicas:**

- 3.1 Listas encadenadas
- 3.2 Árboles binarios
- 3.3 Pilas
- 3.4 Colas

**Palabras clave del capítulo 13**

abstracción de los datos	lista encadenada
apuntador	<i>new</i>
árbol	nodo
árbol binario	nodo termina
árbol de búsqueda binaria	pila
arista	postorden
asignación dinámica	preorden
cola	raíz
conjunto	recorrido
conjunto universal	subárbol
conjunto vacío	subconjunto
diferencia	tipo base
<i>dispose</i>	unión
enorden	variable de apuntador
estructura de datos dinámica	variable de referencia
estructura de datos estática	variable dinámica
intersección	

**EJERCICIOS DEL CAPÍTULO 13**

**★ EJERCICIOS ESENCIALES**

- 1 ¿Cuáles de los siguientes constructores de conjunto son válidos? En el caso de los que son válidos, determíñese el número de miembros del conjunto que se construye.
- a) [1, 1 + 2, 1 + 3, 1 + 4]
  - b) [1..5, 2..6]
  - c) ['A', succ('A'), pred('A')]
  - d) [Lunes..Viernes] si apareció antes la definición de tipo

## TYPE

día = (Lunes, Martes, Miércoles, Jueves, Viernes);

- e) [1.3, 2.7]
  - f) [sqr (2)..sqr(3)]
  - g) [abs(x)..(x)] si  $x$  es una variable entera que contiene 4
  - h) [abs(x)..(x)] si  $x$  es una variable entera que contiene -4
- 2 Describábase un algoritmo para calcular la intersección de dos conjuntos que utilice únicamente la unión de conjuntos y la prueba de membresía.
- 3 Describábase un algoritmo para determinar la cardinalidad, o número de elementos, de un conjunto.
- 4 ¿Por qué es posible asignar NIL a una variable de apuntador independiente mente del tipo de apuntador? ¿No viola esto las reglas de tipo estricto de Pascal? ¿Qué indica esto acerca de la probable representación de un apuntador de valor NIL? ¿Podría sobrevivir Pascal sin el apuntador NIL? En caso afirmativo, ¿qué se usaría para sustituir al apuntador nulo?

### ★ ★ EJERCICIOS IMPORTANTES

- 5 ¿Por qué es razonable que Pascal prohíba el uso de los operadores  $<$ ,  $<=$ ,  $>$ , y  $>=$  para comparar dos apuntadores? (Sugerencia: recuérdese el contenido de los apuntadores.)
- 6 Se mencionó en el texto que existen seis diferentes órdenes de recorrido en un árbol binario de tres nodos. Dado el árbol de búsqueda binaria creado con los caracteres B, A y C (en ese orden), hágase una lista de estos caracteres en los seis recorridos, e indíquese cuáles de ellos corresponden a los recorridos en preorden, enorden y postorden. ¿Cuántos recorridos distintos son posibles en un árbol binario que contiene  $n$  nodos?
- 7 Describábase un algoritmo que invierta el orden de los elementos de una pila. Es decir, si una pila contiene los elementos A, B, C, X, Y y Z, con A en la parte alta de la pila, escribábase un algoritmo que deje a la pila con los elementos Z, Y, X, C, B y A, con Z en la parte alta.
- 8 Defínase la altura de un árbol binario como el número de nodos que se encuentran en el trayecto más largo entre el nodo raíz y un nodo terminal, incluyendo la raíz y el nodo terminal. Describábase un algoritmo para determinar la altura de un árbol binario arbitrario.
- 9 Describábase un algoritmo no recursivo que realice el recorrido enorden de un árbol binario.

### ★ ★ ★ EJERCICIOS ESTIMULANTES

- 10 Dadas las definiciones de tipo

## TYPE

```

apunodo = ↑nodo;
nodo = RECORD
    datos: integer;
    siguiente: apunodo
END;
```

y una lista encadenada cuyos nodos son del tipo nodo, describanse los algoritmos que permitan tener acceso a los elementos de la lista encadenada como si ésta fuera un arreglo. De manera específica, describase un algoritmo para obtener el campo de datos del elemento número  $i$  de la lista encadenada y otro algoritmo para almacenar un nuevo valor en el elemento número  $i$  de la lista encadenada, dado el apuntador que apunta al primer elemento de la lista encadenada, el subíndice del elemento solicitado y el valor nuevo como parámetros. No debe olvidarse el problema de almacenar un valor en un elemento nuevo del arreglo, posiblemente uno que no sea contiguo a los elementos existentes.

- 11** Desafortunadamente, Pascal no cuenta con un mecanismo para manejar contingencias o errores que se pudieran presentar durante la ejecución. Muchos de éstos, como la división entre cero o los intentos por tener acceso a elementos que no están en un arreglo, se pueden evitar si se llevan a cabo las pruebas adecuadas antes de la operación que podría producir la contingencia. En cambio, el procedimiento estándar *new* puede fallar cuando se agota la memoria disponible en el sistema de cómputo y no hay medios para anticipar o manejar este error, ya el tamaño de la memoria disponible varía en gran medida en los diferentes sistemas de cómputo. Describase por lo menos una técnica (de preferencia más de una) para aliviar esta situación. Estímese el impacto de esa técnica sobre programas en Pascal existentes que utilicen el procedimiento estándar *new*.

## PROBLEMAS DEL CAPÍTULO 13 PARA RESOLUCIÓN EN COMPUTADORA

### ★PROBLEMAS ESENCIALES

- 1** Los datos de entrada constan de una lista de enteros terminada por el fin de archivo. Constrúyanse dos listas encadenadas que contengan a estos enteros, una con los enteros en el mismo orden en que aparecen en los datos de entrada y otra con los enteros en orden ascendente. Exhiban después los enteros que aparezcan en la misma posición en ambas listas.

Ejemplo de entrada:

4    3    7    5    9    6    1

Ejemplo de salida

3  
5

- 2** En este problema, el lector debe escribir una simulación para demostrar las operaciones de introducir y sacar elementos de una pila. Cada una de las líneas de datos de entrada incluirá la palabra INTR, SACA o ABRE en las primeras cuatro posiciones. Las líneas que contengan INTR incluirán además un valor

entero. Para cada línea, léase la palabra (y el entero en el caso de las operaciones de introducir), exhibase la operación que se realiza y efectúense después las acciones que se indican en seguida.

- INTR** Introducir el valor entero a la pila.  
**SACA** Sacar un entero de la pila y exhibirlo. Si la pila está vacía, indicarlo.  
**ABRE** Exhibir el contenido actual de la pila, indicando la posición de cada elemento en la pila. Utilíicense únicamente operaciones de introducir y sacar para exhibir estos valores.

Ejemplo de entrada:

```
INTR 4
INTR 5
ABRE
SACA
INTR 2
SACA
SACA
SACA
```

Ejemplo de salida:

```
INTR 4
INTR 5
Apertura de la pila
(partе alta)   1   5
                2   4
SACA
Salió 5 de la pila
INTR 2
SACA
Salió 2 de la pila
SACA
Salió 4 de la pila
SACA
No se puede, la pila está vacía
```

- 3 Cada una de las líneas de datos de entrada contiene entre 1 y 20 enteros en la escala de 1 a 63; el fin de línea seguirá inmediatamente al último entero de cada línea. Constrúyase un árbol de búsqueda binaria que contenga dos valores (datos) en cada nodo. El primer dato representará el número de enteros únicos en una línea de datos de entrada y se usará para ordenar los nodos del árbol. El segundo dato es el conjunto que contiene los enteros de esa línea de datos de entrada. Al construir el árbol de búsqueda binaria, trátense los nodos nuevos que contienen el mismo número de enteros que un nodo ya existente como mayores que el nodo existente. Es decir, se deben insertar a la derecha del nodo existente. Al llegar al fin de archivo, realícese un recorrido

en orden del árbol y exhibábanse los enteros del conjunto de cada nodos en un renglón aparte. Sepárense los enteros en cada renglón mediante una coma y un espacio en blanco.

Ejemplo de entrada:

4	7	1	9
3			
16	3	9	3
21	5	9	4
			5

Ejemplo de salida:

3  
3, 9, 16  
1, 4, 7, 9  
4, 5, 9, 21

## ★ ★ PROBLEMAS IMPORTANTES

- 4 Los datos de entrada constan de dos columnas de palabras (es decir, cada línea contiene dos palabras). Cada una de las palabras estará delimitada por apóstofos y no tendrá más de diez caracteres alfabéticos (aunque pueden aparecer tanto mayúsculas como minúsculas). Las palabras de cada columna están en orden alfabético. Constrúyanse dos lista encadenadas, una con cada columna de palabras, y después fusiónense las dos listas para formar una tercera lista que incluya todas las palabras en orden alfabético. Exhibase la lista que resulta.

Ejemplo de entrada:

'Almendras'	'Arreglos'
'Bananas'	'Buffers'
'Cocos'	'Dígitos'
'Dátiles'	'Programas'

Ejemplo de salida:

Almendras  
Arreglos  
Bananas  
Buffers  
Cocos  
Dátiles  
Dígitos  
Programas

- 5 Los datos de entrada de este problema constan de varias definiciones, organizadas en forma de grupo de líneas de texto. La primera línea de cada definición comienza en la columna 1 con la palabra que se va a definir (de 10 caracteres o menos), por lo menos un espacio en blanco y las primeras palabras de la definición. Las líneas adicionales de cada definición siempre tendrá un espacio en la columna 1 seguido de palabras adicionales de la definición. Constrúyase un árbol binario de estas definiciones y exhibanse después en orden alfabético las palabras definidas. Cada uno de los nodos del árbol binario deberá tener una palabra que se define (y que se emplea para ordenar los nodos del árbol) y un apuntador a una lista encadenada de las palabras que forman la definición.

Ejemplo de entrada:

Nodo	Conjunto de datos empleado para representar los datos y las aristas de un árbol.
Binario	Que tiene dos partes, como en los árboles.
Enlace	Lo que une partes separadas, como los enlaces entre nodos de un árbol binario.

Ejemplo de salida:

Binario	Que tiene dos partes, como en los árboles.
Enlace	Lo que une partes separadas, como los enlaces entre nodos de un árbol binario.
Nodo	Conjunto de datos empleado para representar los datos y las aristas de un árbol.

### ★ ★ ★ PROBLEMAS ESTIMULANTES

- 6 Los datos de entrada de este problema constan de un solo entero,  $N$ . Genérense  $N$  números reales aleatorios únicos entre 0.0 y 1.0 usando cualquier generador de números aleatorios de que se disponga y constrúyase un árbol de búsqueda binaria con los números resultantes. Exhibanse el mayor y el menor de los números generados. Después realícese un recorrido en orden del árbol y calcúlese la diferencia promedio entre números aleatorios adyacentes. En el ejemplo que se muestra a continuación, supóngase que los números aleatorios que se generan son 0.4, 0.2, 0.7, 0.3 y 0.1. Estos números se han limitado a un decimal para simplificar el ejemplo. El cálculo de la diferencia promedio es

$$((0.2 - 0.1) + (0.3 - 0.2) + (0.4 - 0.3) + (0.7 - 0.4))/4$$

(Nota: si  $N$  es demasiado grande, o el generador de números aleatorios se repite después de unos cuantos valores, el programa puede repetir números. Piénsese en un plan para detectar este problema.)

Ejemplo de salida:

El número aleatorio más pequeño es 0.1

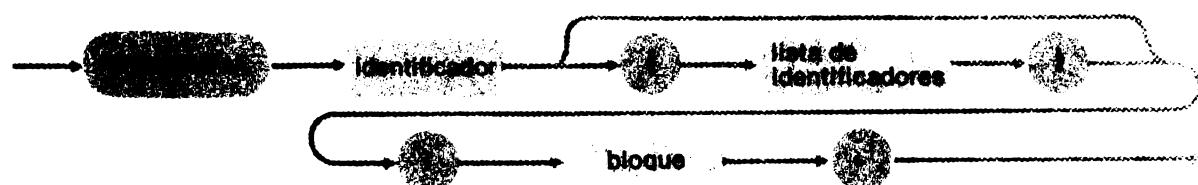
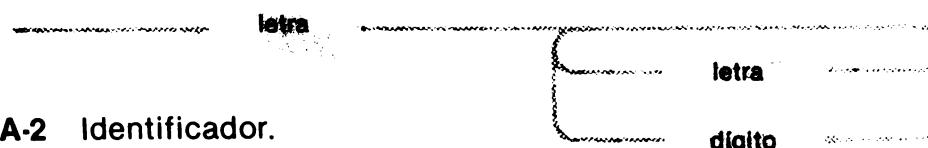
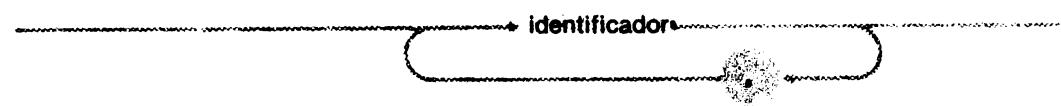
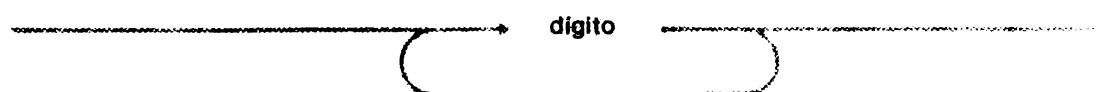
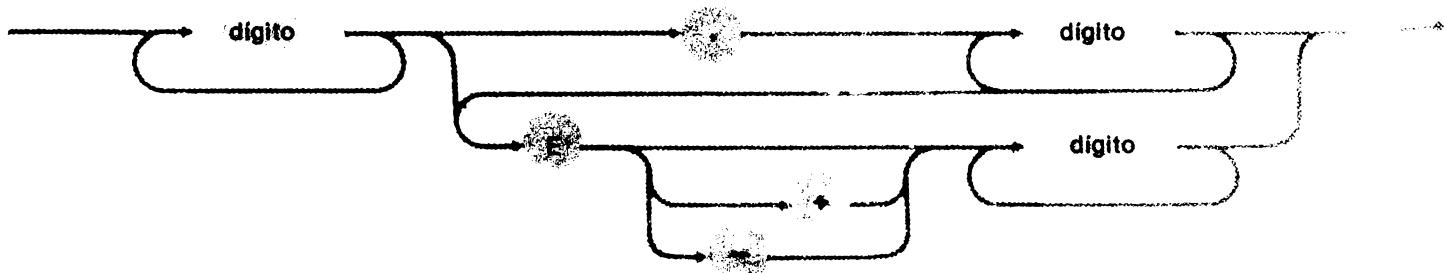
El número aleatorio más grande es 0.7

La diferencia promedio entre números adyacentes es 0.15



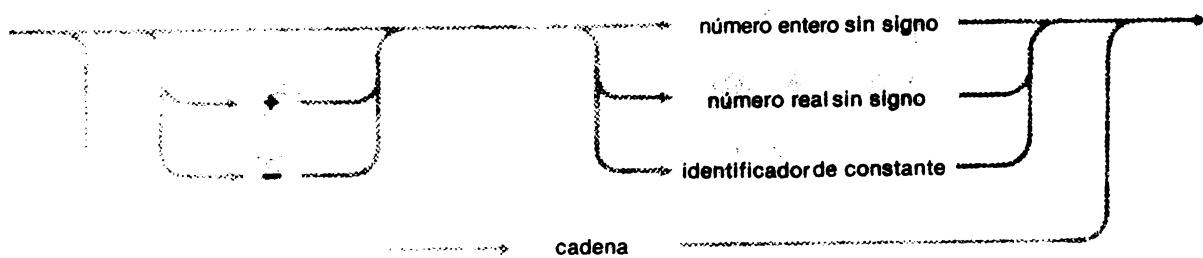
# APÉNDICES

---

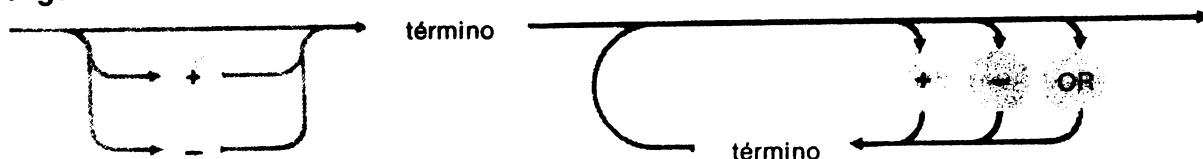
**Figura A-1** Programa.**Figura A-2** Identificador.**Figura A-3** Lista de identificadores.**Figura A-4** Número entero sin signo.**Figura A-5** Número real sin signo.

Fuente: *Programming with Pascal* por B. S. Gottfried (Schaum's Outline Series, McGraw-Hill).

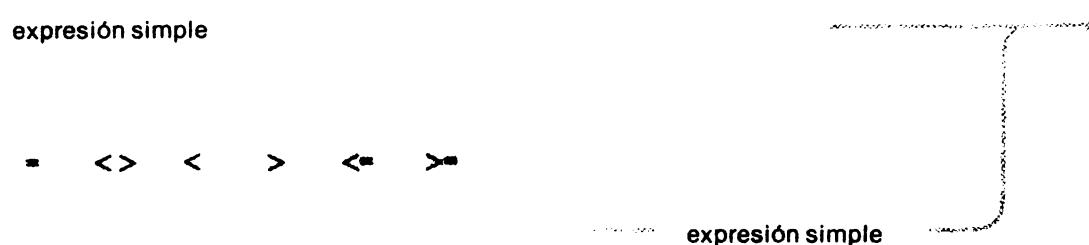
**Figura A-6** Cadena.



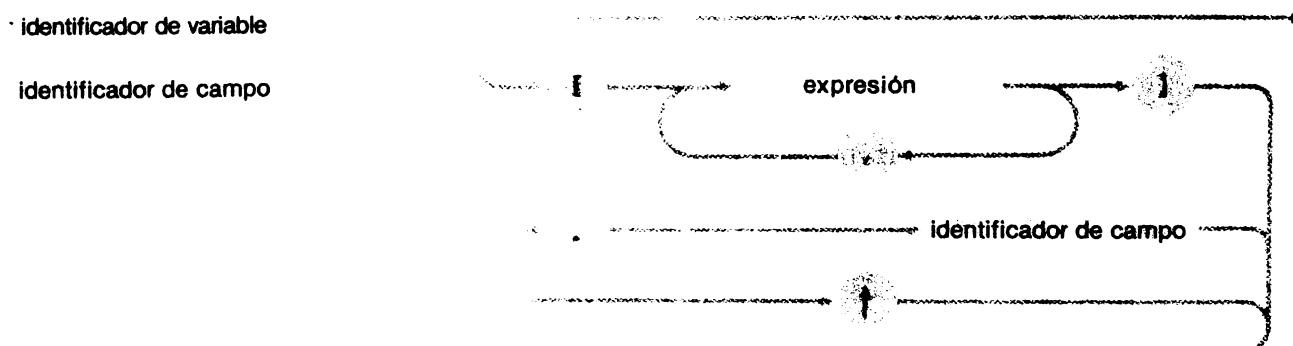
**Figura A-7** Constante.



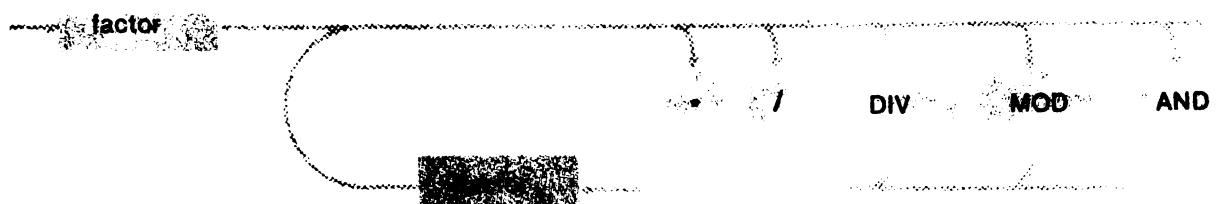
**Figura A-8** Expresión simple.



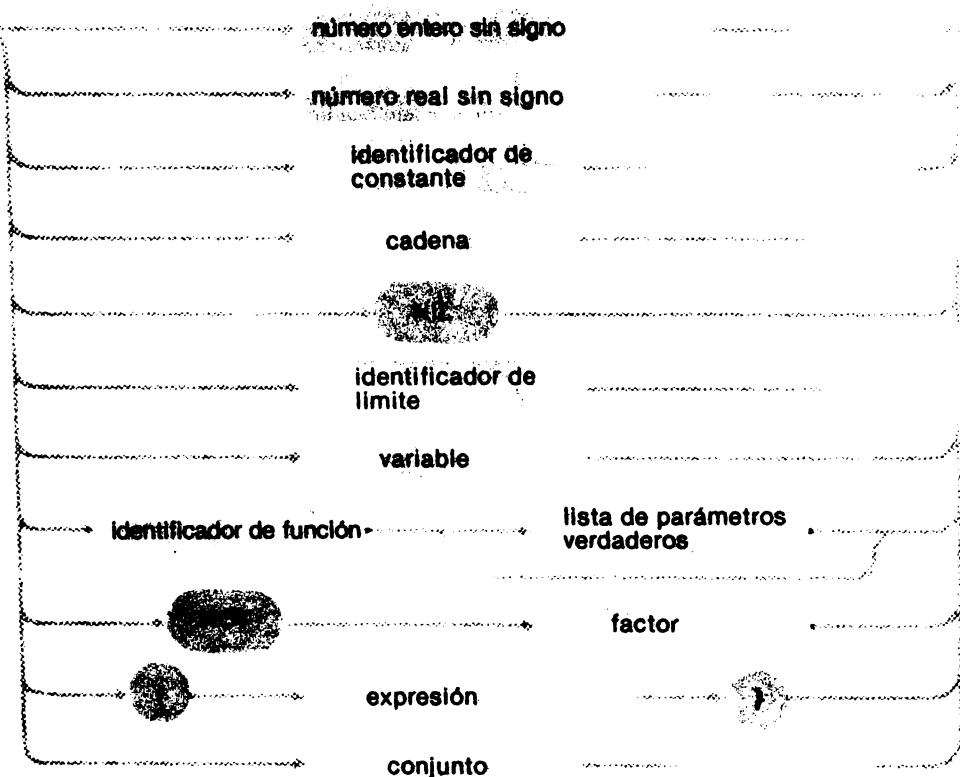
**Figura A-9** Expresión.



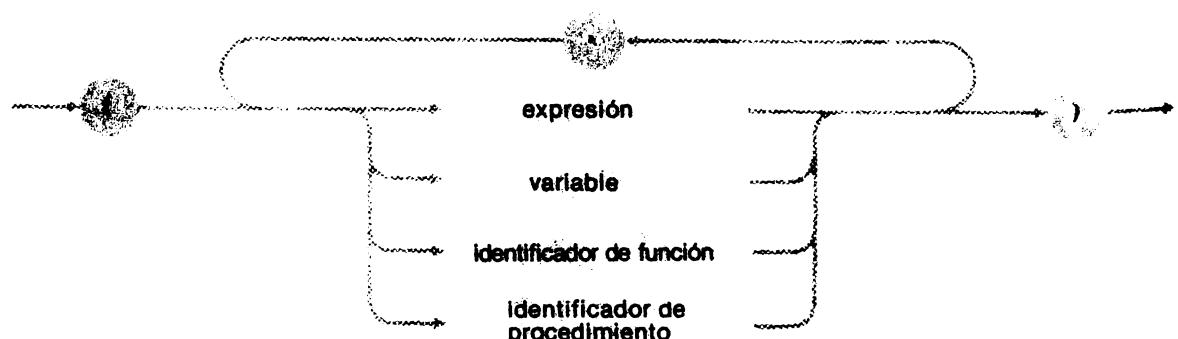
**Figura A-10** Variable.



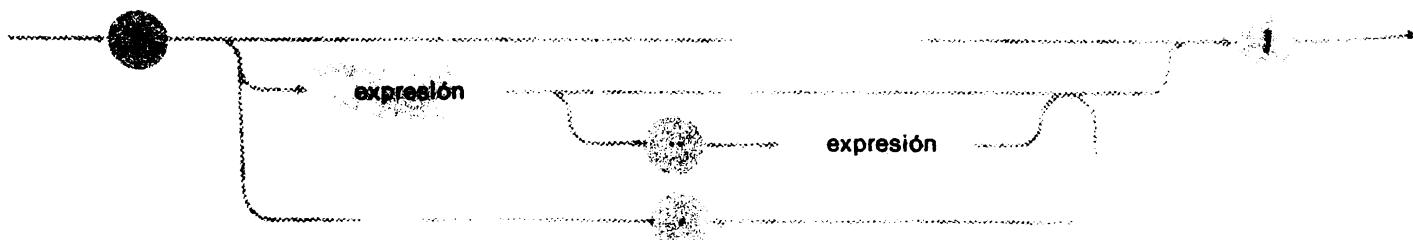
**Figura A-11** Término.



**Figura A-12** Factor.



**Figura A-13** Lista de parámetros verdaderos.



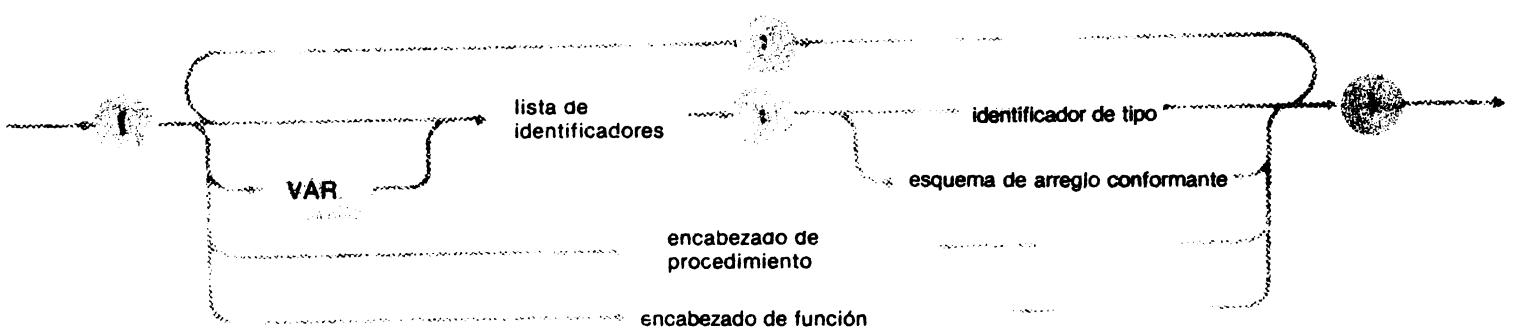
**Figura A-14** Conjunto.



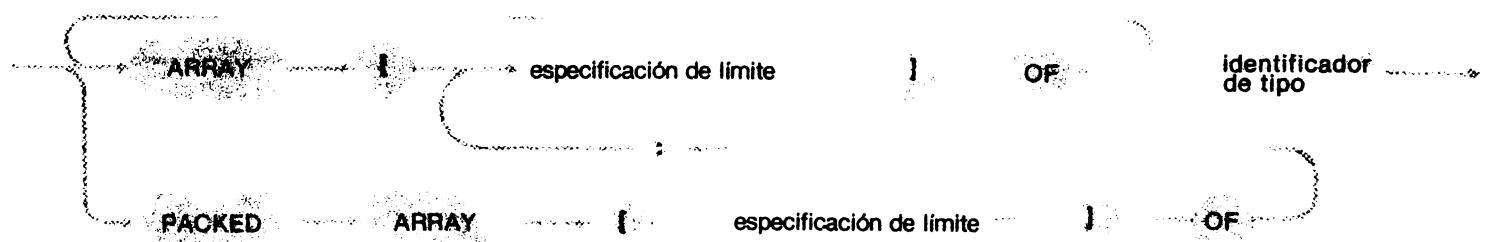
**Figura A-15** Encabezado de función.



**Figura A-16** Encabezado de procedimiento.



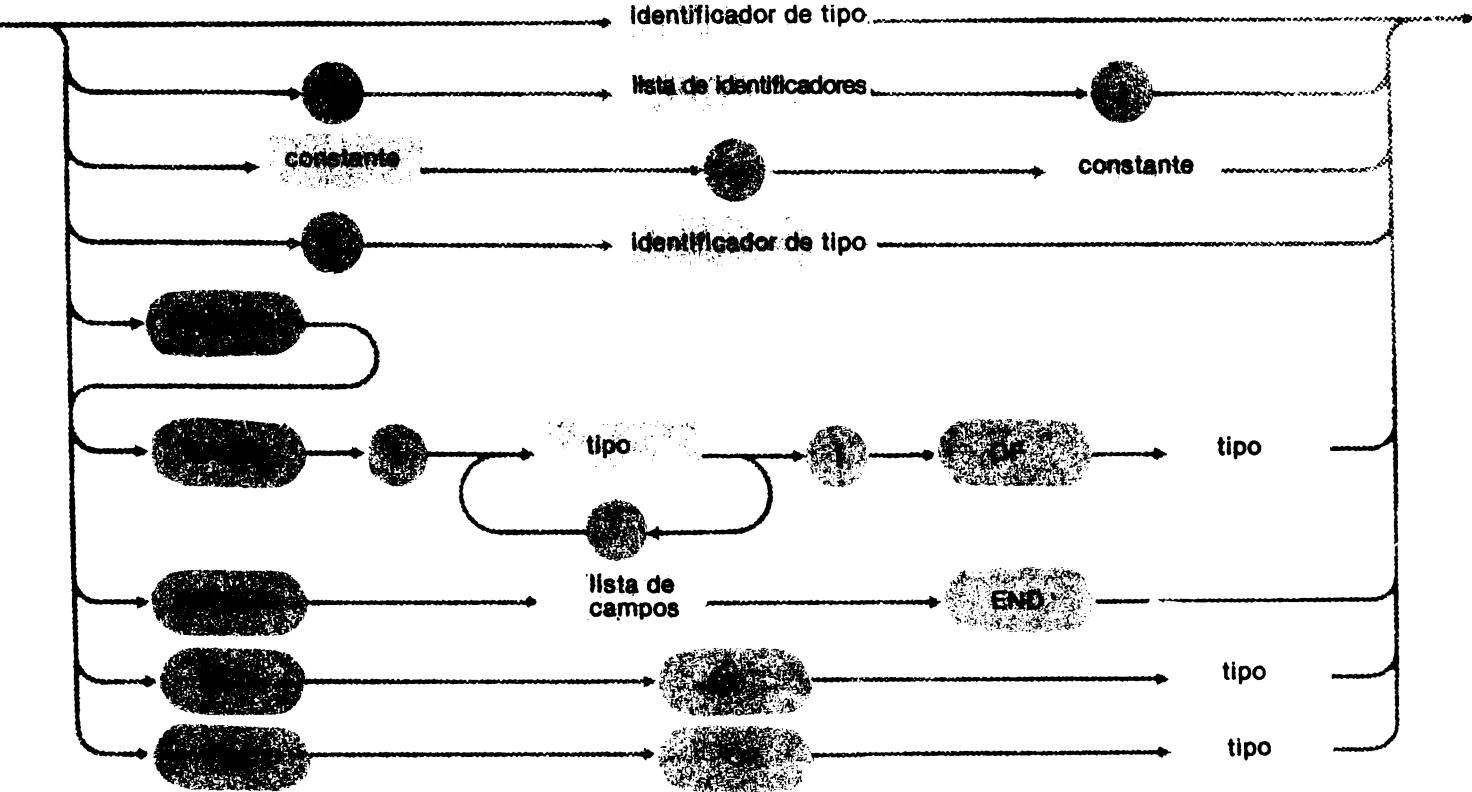
**Figura A-17** Lista de parámetros formales.



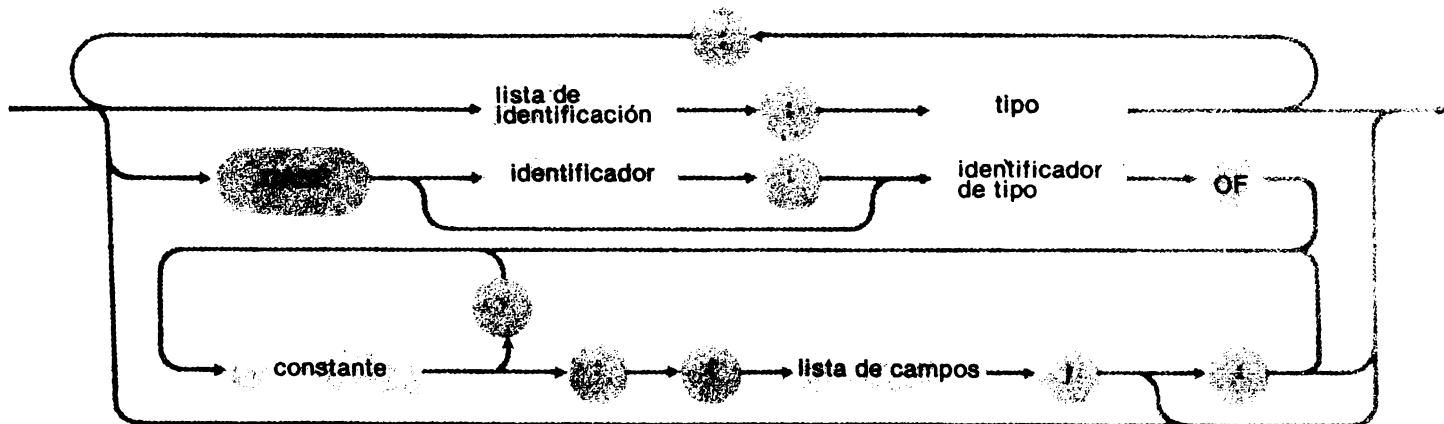
**Figura A-18** Esquema de arreglo conformante.



**Figura A-19** Especificación de límite.



**Figura A-20** Tipo.



**Figura A-21** Lista de campos.

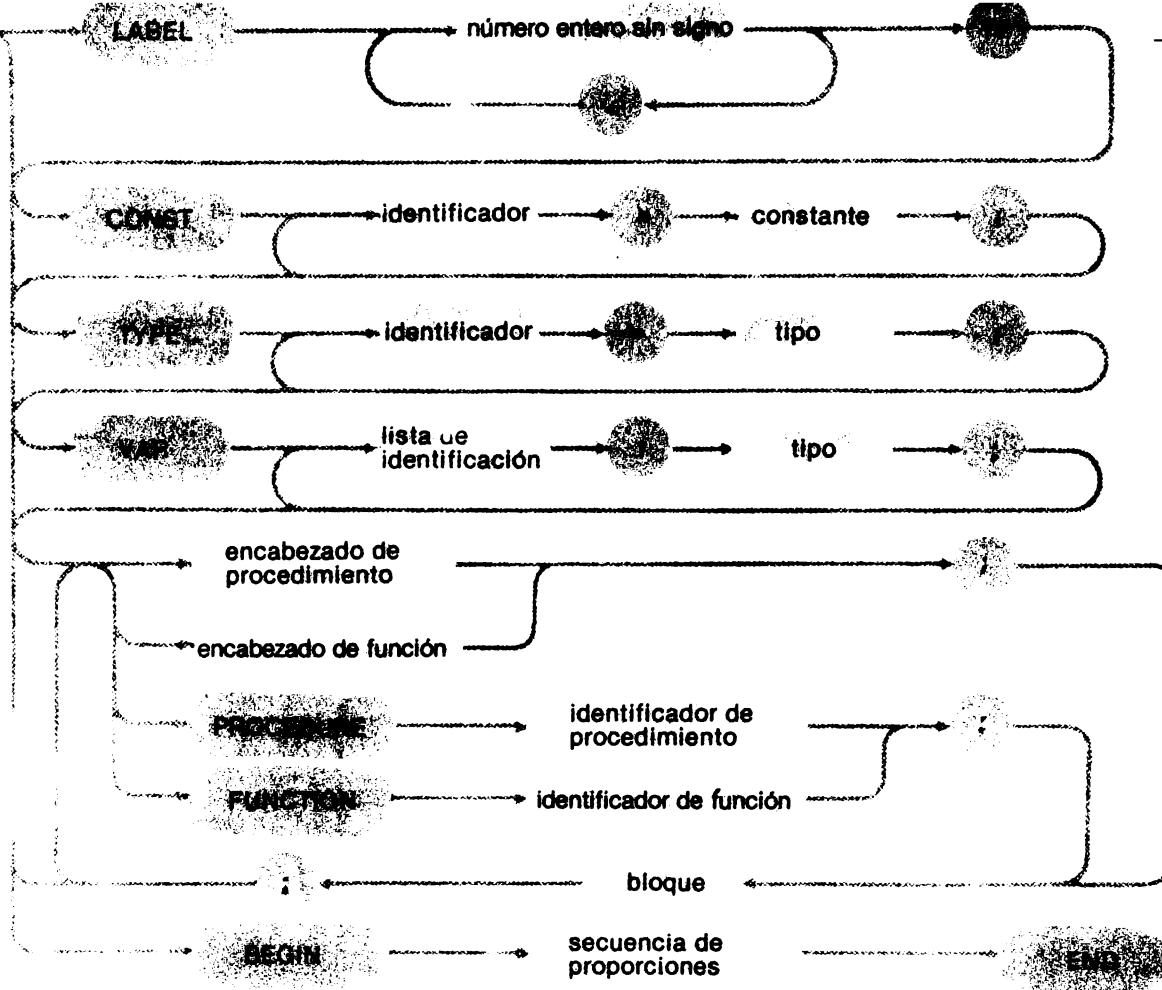
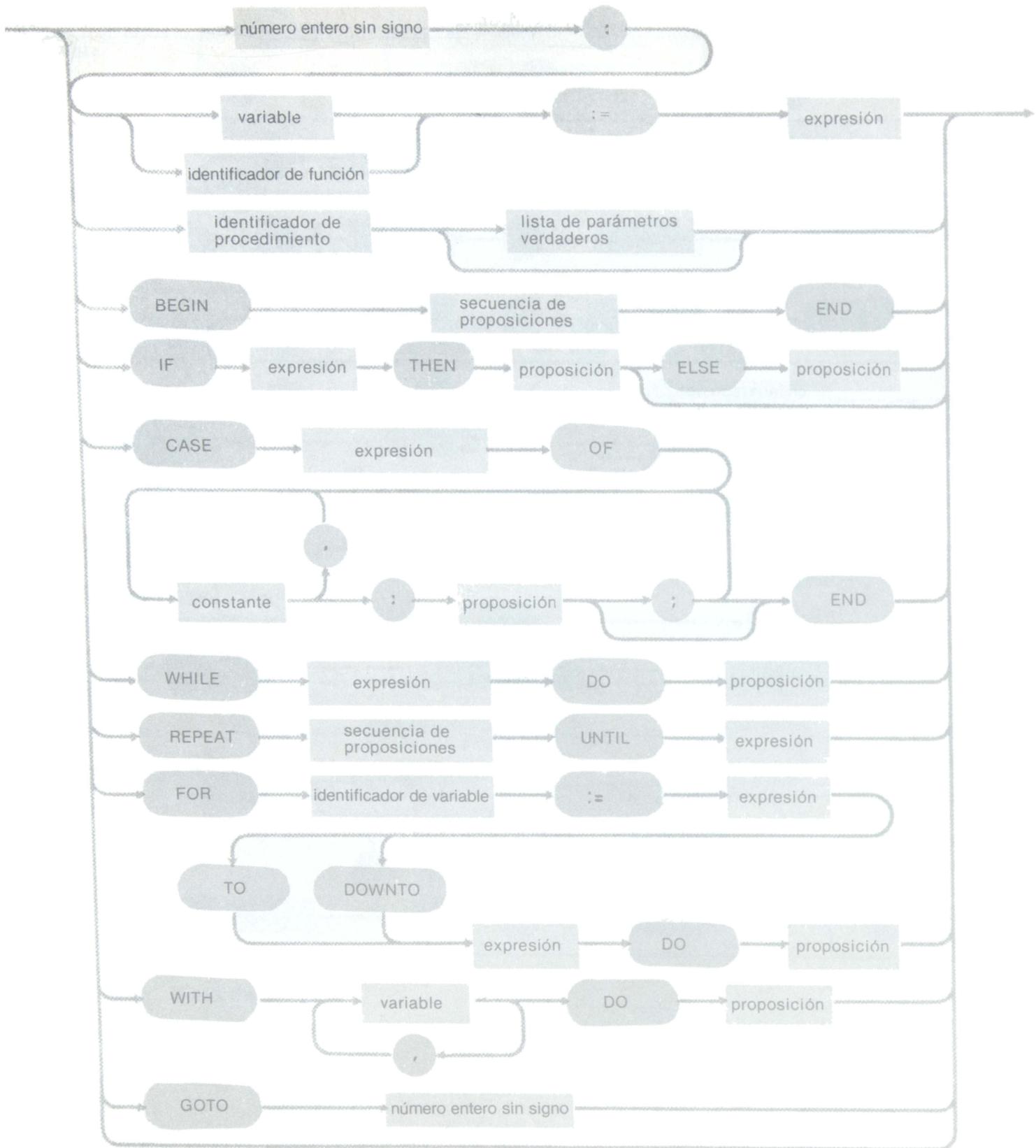


Figura A-22 Bloque.



Figura A-23 Secuencia de proposiciones.



**Figura A-24 Proposición.**

---

AND	END	NIL	SET
ARRAY	FILE	NOT	THEN
BEGIN	FOR	OF	TO
CASE	FUNCTION	OR	TYPE
CONST	GOTO	PACKED	UNTIL
DIV	IF	PROCEDURE	VAR
DO	IN	PROGRAM	WHILE
DOWNTO	LABEL	RECORD	WITH
ELSE	MOD	REPEAT	

---

## APÉNDICE C IDENTIFICADORES PREDECLARADOS

### IDENTIFICADORES ESTÁNDAR

abs	false	pack	sin
arctan	get	page	sqr
boolean	input	pred	sqrt
char	integer	put	succ
chr	ln	read	text
cos	maxint	readln	true
dispose	new	real	trunc
eof	odd	reset	unpack
eoln	ord	rewrite	write
exp	output	round	writeln

### CONSTANTES ESTÁNDAR

false	true	maxint
-------	------	--------

### TIPOS ESTÁNDAR

Boolean	integer	real	text
char			

### ARCHIVOS ESTÁNDAR

input	output
-------	--------

## APÉNDICE D FUNCIONES Y PROCEDIMIENTOS ESTÁNDAR

<i>función</i>	<i>definición</i>	<i>tipo de parametro (x)</i>	<i>tipo de resultado</i>
<b>abs(x)</b>	Valor absoluto de $x$	Entero o real	Mismo que $x$
<b>arctan(x)</b>	Arcotangente de $x$	Entero o real	Real
<b>chr(x)</b>	Carácter que representa el número ordinal $x$	Entero	De caracteres
<b>cos(x)</b>	Coseno de $x$ ( $x$ en radianes)	Entero o real	Real
<b>eof(x)</b>	Probar si se detectó un fin de archivo	Archivo	Booleano
<b>eoln(x)</b>	Probar si se detectó un fin de línea	Archivo	Booleano
<b>exp(x)</b>	$e^x$ , donde $e = 2.71828\ldots$	Entero o real	Real
<b>ln(x)</b>	logaritmo natural de $x$	Entero o real	Real
<b>odd(x)</b>	Probar si $x$ es non o par	Entero	Booleano
<b>ord(x)</b>	Número ordinal de $x$	Cualquier ordinal	Entero
<b>pred(x)</b>	El predecesor de $x$	Cualquier ordinal	Mismo que $x$
<b>round(x)</b>	Redondear el valor de $x$ al entero más cercano	Real	Entero
<b>sin(x)</b>	Seno de $x$ ( $x$ en radianes)	Entero o real	Real
<b>sqr(x)</b>	El cuadro de $x$	Entero o real	Mismo que $x$
<b>sqrt(x)</b>	Raíz cuadrada de $x$ ( $x > 0$ )	Entero o real	Real
<b>succ(x)</b>	El sucesor de $x$	Cualquier ordinal	Mismo que $x$
<b>trunc(x)</b>	Truncar $x$	Real	Entero

## PROCEDIMIENTOS ESTÁNDAR

<i>Procedimiento</i>	<i>Definición</i>
<b>dispose(p)</b>	Indica que la dirección de memoria a la que apunta $p$ está disponible para ser reasignada; $p$ queda sin definir lógicamente.
<b>get(f)</b>	Avanza la variable de archivo $a$ para que el siguiente componente de $a$ quede disponible en $a \uparrow$ .
<b>new(p)</b>	Asignar dinámicamente memoria a una variable del tipo al que apunta $p$ y asignar a $p$ la dirección de esta dirección de memoria.

<i>Procedimiento</i>	<i>Definición</i>
<code>pack(u,i,p)</code>	copiar $a[i], a[i + 1], \dots, a[p]$ comenzando con el primer elemento de $p$ .
<code>page(f)</code>	Agregar fin de línea al archivo de texto $a$ si es necesario, y hacer que el siguiente <i>put</i> , <i>write</i> o <i>writeln</i> que haga referencia a $a$ continúe la salida en una página nueva.
<code>put(f)</code>	Agregar $a \uparrow$ como nuevo componente de $a$ ; $a \uparrow$ queda sin definir.
<code>read(f,lista)</code>	Leer y posiblemente realizar conversión de datos de texto de la variable de archivo $a$ , asignando valores a las variables mencionadas en la lista.
<code>readln(f, lista)</code>	Igual que <i>read</i> , excepto que avanza hasta después del siguiente fin de línea de los archivos de texto como última operación.
<code>reset(f)</code>	Preparar la lectura de la variable de archivo $a$ . Pone a $a$ en modo de entrada o inspección.
<code>rewrite(f)</code>	Preparar la grabación o exhibición en la variable de archivo $a$ ; pone a $a$ en modo de salida.
<code>unpack(p,u,i)</code>	Copiar elementos de $p$ comenzando por el primer elemento a $u[i], u[i + 1], \dots$
<code>write(f, lista)</code>	Grabar, posiblemente convirtiendo a modo textual, el valor de todas las expresiones de la lista en el archivo $a$ .
<code>writeln(f, lista)</code>	Igual que <i>write</i> , sólo que agrega el fin de línea a los archivos de texto como última operación.

## APÉNDICE E OPERADORES

La siguiente tabla resume los diversos operadores de que se dispone en Pascal. Se muestra el orden de prioridad desde la más alta hasta la más baja. Los operadores sucesivos dentro del mismo grupo de prioridad se aplican de izquierda a derecha. Se pueden usar paréntesis para forzar otro orden de evaluación.

<i>prioridad</i>	<i>operador(es)</i>
1 (más alta)	NOT
2	* / DIV MOD AND
3	+ - OR
4 (más baja)	= <> < <= > >= IN

## APÉNDICE F

## ASCII (CÓDIGO AMERICANO ESTÁNDAR PARA INTERCAMBIO DE INFORMACIÓN)

<i>último dígito</i> <i>primer(os) dígito(s)</i>	0	1	2	3	4	5	6	7	8	9
	3	4	5	6	7	8	9	0	1	;
3	!	"	#	\$	%	&	'			
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	C
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	~	^	`	a	b	c
10	d	e	f	g	h	i	j	k	l	ñ
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~			

Los códigos del 0 al 31 representan caracteres especiales de control y no se pueden exhibir. El código 32 representa el espacio en blanco.

## EBCDIC (CÓDIGO DECIMAL CODIFICADO EN BINARIO DE INTERCAMBIO AMPLIADO)

<i>último dígito</i> <i>primer(os) dígito(s)</i>	0	1	2	3	4	5	6	7	8	9
	7	8	9	0	1	2	3	4	5	6
7	&									
8										
9	!	\$	*	)	;	-	-	/	:	'
10								,	%	-
11	>	?								
12			:		#	@	'	=	"	
13	b	c	d	e	f	g	h	i		
14						j	k	l	m	
15	o	p	q	r	u	v	w	x	y	
16			s	t						
17										
18	[	]								
19					A	B	C	D	E	F
20	H	I								
21	K	L	M	N	O	P	Q	R	T	U
22										
23	W	X	Y	Z						
24	0	1	2	3	4	5	6	7	8	'

Los códigos del 0 al 63 y del 250 al 255 representan caracteres especiales de control y no se pueden exhibir. El código 256 representa el espacio en blanco.

## EL PROCEDIMIENTO PAGE

Pascal incluye un procedimiento estándar llamado *page* que hace que el siguiente elemento de salida se exhiba en la parte superior de una página nueva. Considerese el siguiente segmento de código:

```
writeln (2,4);  
page;  
writeln (6,8)
```

Los valores enteros 2 y 4 se exhiben en un renglón. En seguida, el dispositivo de salida se posiciona de tal modo que los siguientes caracteres se exhiban en una página nueva. Por último se exhiben los valores 6 y 8 en la página nueva.

El procedimiento *page* depende del sistema usado. Por ejemplo, en algunos sistemas interactivos se hace caso omiso del procedimiento *page* cuando la salida se exhibe en una pantalla de terminal, pero en otros sistemas interactivos se borra la pantalla de la terminal. Si el dispositivo de salida es una impresora, el procedimiento *page* probablemente hará que la impresora avance el papel hasta que llegue a la parte superior de la siguiente página. Si el lector se propone utilizar el procedimiento *page*, deberá asegurarse de leer la documentación de su sistema para verificar el efecto que tiene sobre el dispositivo de salida.

## LA PROPOSICIÓN GOTO

Muchos lenguajes de programación, como FORTRAN y BASIC, incluyen una estructura de control que realiza una transferencia incondicional; es decir, el control se transfiere incondicionalmente de una proposición a otra proposición que esté etiquetada, normalmente mediante un número entero.

En Pascal sí existe una estructura de control de este tipo y se llama *proposición GOTO* (ir a). La proposición consta de la palabra reservada GOTO seguida de un entero en la escala de 1 a 9999; este entero se llama *etiqueta*. El control se transfiere de manera incondicional a la proposición dentro del mismo procedimiento, función o programa principal, que vaya precedida por la etiqueta especificada y un signo de dos puntos.

Todas las etiquetas empleadas en un procedimiento, función o programa principal deben declararse antes de las demás declaraciones. El siguiente procedimiento ilustra el uso de la proposición GOTO y la declaración de etiquetas; lee cien valores y da como resultado el total.

```
PROCEDURE sumadatos (VAR total: integer);  
(* Calcular el total de cien valores enteros capturados. *)  
LABEL  
    10;  
CONST  
    máx = 100;  
VAR
```

**BEGIN**

```

    total := 0;
    índice := 1;
10 : read (núm);
    Total := total + núm;
    índice := índice + 1;
    IF índice > 101
END;
```

Nótese que la palabra reservada LABEL va seguida de uno o más enteros separados por comas y que declaran a las etiquetas. Se hace caso omiso de las etiquetas durante la ejecución, excepto cuando una proposición GOTO hace referencia a una de ellas. El mismo procedimiento se puede escribir con cualquiera de las estructuras de control cíclicas que se analizaron en este texto (WHILE, REPEAT-UNTIL o FOR). Se ha evitado el uso de proposiciones GOTO en los programas. El código que se escribe usando exclusivamente estructuras de control diferentes de la proposición GOTO tiene únicamente un punto de entrada y un punto de salida, mientras que el uso de GOTO permite varios métodos diferentes de entrada y salida del código. Por tanto, el empleo de la proposición GOTO puede producir programas que son difíciles de leer, comprender y depurar.

Las proposiciones GOTO no se pueden usar para saltar a una proposición que sea parte de otra estructura de control, a menos que, por supuesto, la proposición GOTO misma sea parte de esa estructura. Existen también situaciones en las que se pueden usar la proposición GOTO para terminar la ejecución de procedimientos y funciones. Por fortuna, tales aplicaciones no son frecuentes.

Es probable que el lector se pregunte si existen situaciones en las que sea adecuado el uso de la proposición GOTO. En aquellas situaciones especiales en las que se desea terminar de inmediato un procedimiento, función o programa a causa de un error importante, es aceptable saltar directamente al final de las proposiciones ejecutables. No obstante, aun esta aplicación del GOTO es questionable, por lo que se recomienda al lector evitar esta proposición siempre que sea posible.

## **PROCEDIMIENTOS Y FUNCIONES COMO PARÁMETROS**

Si el lector examina los diagramas de sintaxis (apéndice A) que corresponden a la lista de parámetros verdaderos y la lista de parámetros formales, descubrirá que los procedimientos y funciones pueden aparecer como parámetros. Así, por ejemplo, un parámetro verdadero de una invocación de procedimiento puede ser un procedimiento o función. El siguiente encabezado de procedimiento incluye una función como parámetro formal:

```

PROCEDURE proceso (FUNCTION f (x : integer) : real;
                   izq, der : integer);
```

Adviértase que, si el parámetro formal es una función (o procedimiento), se escribirá de la misma forma como se escribe el encabezado de la función (o procedimiento).

El parámetro verdadero de función (o procedimiento) que se pasa al procedimiento es únicamente el nombre de la función. Por ejemplo, las siguientes invocaciones de procedimiento son válidas (si *func1* y *func2* son funciones de valor real, cada una de las cuales con un parámetro formal de valor de tipo entero).

```
proceso (func1, 0, 5)  
proceso (func2, 1, 4)
```

Existen restricciones respecto a los procedimientos o funciones que se pueden pasar como parámetros. Por ejemplo, los parámetros de una función o procedimiento que se pase como parámetro deben ser parámetros de valor. Algunas versiones de Pascal no permiten usar los procedimientos y funciones estándar como parámetros. El lector deberá estudiar la documentación de su sistema antes de intentar usar procedimientos o funciones como parámetros.



# GLOSARIO

## **Abstracción de los datos**

Proceso de diseño de estructuras de datos, similar al diseño descendente y en el que se posponen los detalles de la estructura de datos real hasta que se necesitan.

## **Alcance**

Parte del programa en la que se conoce un identificador.

## **Algoritmo**

Procedimiento paso por paso para resolver un problema dado y que termina en un número finito de pasos.

## **Almacenamiento secundario**

Dispositivos de memoria que proporcionan espacio de almacenamiento adicional, como las unidades de disco y cinta.

## **Árbol**

Estructura de datos dinámica que contiene un nodo raíz al que están

asociados cero o más nodos llamados descendientes, cada uno de los cuales también tiene cero o más nodos descendientes que también pueden tener descendientes y así sucesivamente.

## **Árbol binario**

Árbol en el que cada nodo tiene cero, uno o dos descendientes.

## **Árbol de búsqueda binaria**

Estructura de datos dinámica en la que los datos se pueden localizar e insertar de manera eficiente.

## **Archivo**

Secuencia de componentes que tienen un nombre y que normalmente se almacena en disco o cinta magnética y al cual se hace referencia en Pascal mediante las proposiciones READ y WRITE.

## **Archivo de acceso aleatorio**

Véase archivo de acceso directo.

**Archivo de acceso directo**

Archivo en el que la recuperación o almacenamiento de un dato no depende del orden de los componentes

Por ejemplo, un dispositivo de acceso directo como los discos.

**Archivo externo**

Archivo permanente utilizado o creado por un programa en Pascal y al cual se hace referencia a través de un parámetro de la proposición PROGRAM.

**Archivo interno**

Archivo cuya variable de archivo correspondiente no aparece en la proposición PROGRAM y cuya duración está limitada a la ejecución de un solo programa.

**Archivo secuencial**

Archivo que requiere que la recuperación o almacenamiento de elementos se lleve a cabo en orden secuencial.

**Argumento**

Valor que se proporciona a una función con el fin de obtener el resultado.

**Arista**

Conexión entre dos nodos de un árbol.

**Arreglo**

Tipo de datos estructurado con un número fijo de componentes del mismo tipo. Se tiene acceso a cada uno de los componentes mediante un subíndice o índice.

**Arreglo clasificado**

Arreglo cuyos elementos están ordenados en una secuencia específica.

**Arreglo de registros**

Arreglo en el que los componentes son registros.

**Arreglo empacado**

Arreglo que puede ahorrar espacio de memoria al empacar varios componentes del arreglo en una sola celda de memoria.

**Arreglo multidimensional**

Arreglo cuyos componentes son arreglos.

**Arreglos paralelos**

Dos o más arreglos con los mismos límites y en los que los elementos que tienen el mismo subíndice se manejan como un solo elemento compuesto. Compárese con registro.

**Aserción**

Comentario acerca del estado del programa durante la ejecución en el punto de la aserción.

**Asignación dinámica**

Proceso de apartar memoria para variables dinámicas durante la ejecución.

**Binario**

Se refiere a dos estados, o base dos.

**Bit**

Contracción de binary digit (dígito binario) que es cero o uno. Los bits son la unidad de información más pequeña que se utiliza en una computadora y sirven para representar datos e instrucciones.

**Bloque de programa**

El programa en Pascal (excluyendo el encabezado) que consta de las declaraciones y proposiciones ejecutables.

**Booleano**

Tipo de datos en Pascal que sólo tiene dos valores posibles: true (verdadero) y false (falso).

**Búsqueda binaria**

Técnica de búsqueda para localizar un elemento de una lista ordenada mediante la reducción repetida del área de búsqueda a la mitad hasta localizar el elemento deseado.

**Búsqueda lineal**

Técnica de búsqueda que localiza un valor especificado mediante su comparación con todos los elementos de una lista hasta que se encuentra el elemento o se termina la lista.

**Búsqueda secuencial**

Véase búsqueda lineal.

**Cadena**

Secuencia de caracteres que se considera como un solo dato, arreglo empacado de caracteres.

**Cadena de caracteres**

Secuencia de caracteres que se manejan lógicamente como una unidad.

**Campo etiqueta**

Sé usa en los registros variantes para determinar la estructura actual del registro.

**Carga**

Véase "enlace".

**Centinela**

Dato especial que sirve para indicar el final de un archivo de datos.

**Char**

Tipo de datos en Pascal que consta de caracteres como los que se encuentran en un teclado o impresora.

**Ciclo infinito**

Ciclo en el que nunca se cumple la condición de terminación y sólo se puede terminar mediante una acción externa al programa.

**Ciclos**

Proceso que permite la ejecución repetida de una secuencia de proposiciones hasta cumplirse una condición de terminación.

**Clasificación de burbuja**

Técnica de clasificación muy conocida que intercambia elementos adyacentes hasta que el valor deseado "sube como burbuja" hasta el extremo de la lista.

**Clasificación por selección**

Algoritmo de clasificación de uso común que localiza el elemento más grande cada vez que examina una lista y lo coloca en la posición correcta.

**Código**

Proposiciones o instrucciones a las que se hace referencia como un grupo.

**Códigos mnemotécnicos**

Códigos simbólicos que se emplean para representar expresiones binarias en lenguaje de máquina.

**Cola**

Estructura de datos dinámica cuyos elementos se introducen en un extremo y se recuperan en el otro. A menudo se le llama cola de primero que entra, primero que sale (en inglés, first-in, first-out o FIFO).

**Columna**

Normalmente se refiere a todos los elementos de un arreglo tridimensional que tienen el mismo valor como segundo subíndice.

**Comentario**

Componente de un programa de computadora que se usa para documentar y explicar partes del programa a los lectores humanos. La computadora normalmente hace caso omiso de los comentarios.

**Compatible para asignación**

Se refiere a la asignación de valores de un tipo a una variable de tipo compatible.

**Compilador**

Programa que traduce un programa escrito en un lenguaje de alto nivel, como el Pascal, en lenguaje de máquina para una computadora específica.

**Concatenación**

Operación que consiste en juntar dos o más datos, como son las cadenas de caracteres, para formar una sola secuencia.

**Conjunto**

Tipo de datos estructurado en Pascal que consta de un grupo de elementos distintos elegidos de entre las constantes del tipo base.

**Conjunto universal**

En Pascal un conjunto formado por todas las constantes del tipo base.

**Conjunto vacío**

El conjunto que no tiene elementos.

**Constante**

Valor que no cambia durante la ejecución de un programa.

**Constructor de conjunto**

Lista explícita de los miembros.

**Declaración**

Expresión del Pascal que define o declara objetos de programación que requiere el programa, como son variables, constantes y procedimientos y funciones.

**Definición de tipo**

En Pascal, la definición de un tipo de datos.

**Delimitador**

Símbolo o cadena de caracteres que delimita o marca una parte de un programa o expresión; por ejemplo, BEGIN y END.

**Depuración**

Proceso de eliminar errores de un programa.

**Diagrama de sintaxis**

Ayuda visual que sirve para expresar reglas de sintaxis.

**Diferencia**

En Pascal, operación de conjuntos para formar la diferencia de dos conjuntos.

**Dirección de memoria**

Número único que se asigna a cada celda de la unidad de memoria. La utiliza la UCP para hacer referencia a celdas de memoria individuales.

**Diseño descendente**

Proceso de dividir un problema en subproblemas y después dividir los subproblemas hasta que se puedan resolver fácilmente.

**Dispose**

Procedimiento del Pascal que libera la memoria apartada para una variable dinámica.

**Dispositivo de entrada**

Dispositivo que permite al usuario introducir datos e información a la computadora; por ejemplo, el teclado.

**Dispositivo de salida**

Dispositivo que exhibe la información que la computadora proporciona al usuario; por ejemplo, una impresora.

**Dispositivo periférico**

Dispositivo de E/S que normalmente es externo a la computadora.

**Divide y vencerás**

Estrategia de resolución de problemas que conlleva la división de un problema en problemas más pequeños y manejables.

**Editor**

Programa interactivo que permite al usuario crear y modificar archivos de texto.

**Efecto secundario**

Resultado de modificar una variable que se declaró fuera de un procedimiento o función y que no se pasó como parámetro.

**Ejecutar**

Llevar a cabo los pasos de un algoritmo o programa en lenguaje de máquina.

**Encabezado de programa**

Una sola proposición en Pascal que comienza con la palabra reservada PROGRAM e incluye el nombre del programa y los nombres de las variables de archivo asociadas a archivos externos.

**Enlace**

Proceso en el que dos o más programas objeto se enlazan con los procedimientos de biblioteca que utilice el sistema.

**Enorden**

Se refiere al algoritmo de recorrido de árboles que recorre primero el subárbol izquierdo, después la raíz y finalmente el subárbol derecho.

**Ensamblador**

Programa que traduce un programa en lenguaje de ensamblaje a lenguaje de máquina.

**Entero**

Tipo de datos del Pascal que consta de números enteros positivos y negativos y el cero.

**Eof**

Fin de archivo.

**Eoln**

Marca de fin de línea que se emplea en archivos de texto.

**Error de compilación**

Error que se detecta durante la compilación de un programa, como los errores de sintaxis.

**Error de ejecución**

Error que se presenta durante la ejecución de un programa de computadora.

**Error de lógica**

Error que generalmente se remonta al diseño del algoritmo.

**Error de semántica**

Error relacionado con el significado de una proposición.

**Errores de programación**

Errores que tienen los programas y que pueden ser errores de sintaxis, de ejecución o de lógica.

**Estructura de control**

Construcción de programación que determina el orden en que se van a ejecutar las proposiciones, como por ejemplo la selección y los ciclos.

**Estructura de control definida**

Estructura de control en la que el número de veces que se repite un ciclo se determina antes de la ejecución, como en el caso del ciclo FOR.

**Estructura de control indefinida**

Estructura de control (por ejemplo, WHILE o REPEAT-UNTIL) en la que el número de iteraciones del cuerpo del ciclo puede variar dependiendo del valor de una expresión booleana arbitraria.

**Estructura de datos dinámica**

Estructura de datos que puede ampliarse o reducirse durante la ejecución; por ejemplo, las listas ligadas.

**Estructura de datos estática**

Estructura de datos cuyo tamaño queda fijado durante la compilación. Por ejemplo, los arreglos.

**Exactitud**

Medida de la calidad de correcto o de la precisión.

**Expresión booleana**

Expresión que, al evaluarse, produce un valor booleano.

**Filtros**

Programas que realizan transformaciones sobre un archivo de entrada para producir un archivo de salida.

**Formato**

Se refiere a la organización, o representación, de la información que se exhibe en un archivo de texto.

**Forward**

Declaración de procedimiento o función de Pascal que aparece antes que su cuerpo.

**Función chr**

Función cuyo resultado es el carácter cuya posición (o código) en la secuencia de colocación es el mismo que el argumento entero de la función.

**Función de transferencia**

Función que convierte de un tipo de datos a otro, como la función round.

**Función definida por el usuario**

Función que define y declara el programador.

**Función eof**

Función del Pascal que sirve para determinar si quedan componentes por leer en un archivo de entrada.

**Función eoln**

Función del Pascal que sirve para determinar si el siguiente componente por leer de un archivo de texto es la marca de fin de línea.

**Función estándar**

Función incluida en el Pascal, como por ejemplo la raíz cuadrada (sqrt).

**Función ODD**

Función estándar del Pascal que sirve para probar si el parámetro entero es par o non.

**Función ORD**

Función estándar del Pascal que sirve para determinar la posición ordinal del parámetro ordinal de entrada.

**Función pred**

Función estándar del Pascal que produce el predecesor del parámetro ordinal de entrada.

**Función recursiva**

Función que se invoca a sí misma, ya sea en forma directa o indirecta.

**Función succ**

Función estándar en Pascal que produce el sucesor del parámetro ordinal de entrada.

**Funciones ordinales estándar**

Las funciones estándar del Pascal succ, pred, ord y chr.

**FUNCTION**

Palabra reservada del Pascal que se usa en la declaración de una función.

**Fusión**

Proceso de combinar dos o mas listas ordenadas para producir una sola lista ordenada.

**Get**

Procedimiento del Pascal que avanza la proposición actual de la ventana de archivo al siguiente componente de un archivo secuencial y asigna el valor del componente a la variable de almacenamiento temporal del archivo.

**Hardware**

Componentes físicos de la computadora, a diferencia del software; incluye los dispositivos de entrada/salida.

**Hilera**

Normalmente se refiere al conjunto de todos los elementos de un arreglo bidimensional que tienen el mismo valor que el primer subíndice.

**Identificador**

Palabra o secuencia de caracteres que constituye el nombre de un objeto de programa, como por ejemplo un nombre de programa, una variable, una constante, un tipo o una función o procedimiento.

**Identificador de campo**

En Pascal, nombre del componente de un registro.

**Identificador estándar**

Identificador en Pascal, como output o input, que declara previamente el compilador y se asocia automáticamente a ciertos objetos de programación.

**Identificador global**

Identificador que se declara en un procedimiento o programa principal y por tanto está accesible en cualquier procedimiento o función dentro del alcance del identificador.

**Identificador local**

Identificador que se conoce solamente dentro de su alcance

**Impresión de eco**

Exhibición o impresión de los datos de una entrada en el momento de captura y que se usa principalmente para validar las entradas.

**Impulsor**

Programa principal cuyas proposiciones son sobre todo invocaciones de procedimientos.

**Indicador**

Variable que se usa para indicar la ocurrencia de un evento o el cumplimiento de una condición. En Pascal se usan a menudo variables booleanas como indicadores.

**Indicador de campo**

En Pascal, expresión que se usa para tener acceso a los componentes de un registro.

**Índice**

Subíndice que se emplea para tener acceso a los componentes de un arreglo.

**Input**

En Pascal, identificador estándar que representa al archivo de entrada estándar usado por omisión en las proposiciones read y readln.

**Intérprete**

Programa que traduce y ejecuta las proposiciones de un programa fuente proposición por proposición.

**Intersección**

Operación de conjuntos representada por \* y que forma el conjunto de elementos comunes a dos o mas conjuntos.

**Invariante de ciclo**

Aserción que no cambia después de cada iteración de un ciclo.

**Invocación**

Transferencia de control de una proposición de procedimiento (o función) al procedimiento (o función) en sí.

**Jerarquía**

Orden o prioridad al evaluar una expresión que incluye más de un operador.

**Lenguaje de alto nivel**

Lenguaje de computadora en el que las instrucciones de cómputo se representan mediante proposiciones que se asemejan a expresiones en inglés.

**Lenguaje de bajo nivel**

Lenguaje que expresa instrucciones de cómputo en forma simbólica íntimamente relacionada con el sistema de cómputo específico que debe ejecutar las instrucciones.

**Lenguaje de ensamblaje**

Lenguaje de computación íntimamente relacionado con el lenguaje de máquina y que utiliza códigos mnemotécnicos para representar instrucciones en lenguaje de máquina.

**Lenguaje de máquina**

Lenguaje que expresa instrucciones de cómputo en código binario.

**Lenguaje de tipos estrictos**

Lenguaje, como el Pascal, en el que todos los identificadores se deben declarar especificando su tipo. Esto permite al compilador verificar antes de la ejecución que los objetos se utilicen únicamente en el contexto de tipo apropiado.

**Lista ligada**

Estructura de datos dinámica que consta de una lista encadenada de datos conectados mediante punteros.

**Llave de registro**

Componente de un registro que se usa como llave, o valor identificador, al procesar los registros.

**Maxint**

Constante entera estándar predefinida que representa al entero más grande que se puede manejar en un sistema de cómputo determinado.

**Memoria primaria**

Unidad principal de memoria (en contraste con almacenamiento secundario).

**Memoria principal**

Memoria primaria que usa el sistema de cómputo para almacenar instrucciones y datos a los que va a tener acceso directo la UCP. También se le conoce como "núcleo".

**Módulo instrumental**

Procedimiento que se usa para la prueba y depuración y que sustituye temporalmente al procedimiento verdadero.

**New**

Procedimiento estándar en Pascal que aparta dinámicamente memoria para una variable.

**Nodo**

Punto o vértice de un árbol. Normalmente representa a un dato.

**Nodo terminal**

Nodo que no tiene descendientes. También se le llama nodo hoja.

**Nombre de archivo**

Nombre que se usa para identificar permanentemente un archivo. Este nombre es por lo común distinto del nombre de la variable de archivo que se usa para tener acceso al archivo.

**Notación científica**

Notación matemática abreviada que usa potencias de diez para representar números muy grandes o muy pequeños. También se conoce como notación de punto flotante.

**Notación de punto flotante**

Llamada también notación científica. Se usa para representar valores reales.

**Notación O**

Sirve para estimar matemáticamente el rendimiento o eficiencia de un algoritmo.

**Número aleatorio**

Número generado al azar.

**Número seudoaleatorio**

Número generado por programa de tal manera que parece ser aleatorio.

**Operador AND**

Operador lógico de Pascal que resulta en el valor true (verdadero) si ambos operandos se cumplen y false (falso) en caso contrario.

**Operador aritmético**

Operador como  $+, -, *, /, \text{DIV}$  y MOD que se usa en cálculos numéricos.

**Operador binario**

Operador que requiere dos operandos, como la suma o la multiplicación.

**Operador booleano**

En Pascal, uno de los operadores AND, OR y NOT.

**Operador IN**

Operador del Pascal que sirve para determinar la membresía en un conjunto.

**Operador NOT**

Operador booleano que produce el valor lógico opuesto del operando.

**Operador OR**

Operador lógico del Pascal que produce el valor true si cualquiera de los operandos vale true y false en caso contrario.

**Operador unario**

Operador que sólo requiere un operando, como NOT.

**Operadores relacionales**

Operadores del Pascal como ? y » que se usan para comparar datos del mismo tipo. Estos operadores producen un valor booleano.

**Operando**

Valor que se usa al evaluar un operador como el de adición (+) o el de multiplicación (\*).

**Orden**

Estimación de la eficiencia de un algoritmo.

**Orden lexicográfico**

Orden "real" de un tipo de datos ordinal. Por ejemplo, orden alfabetico.

**Pack**

Procedimiento estándar del Pascal que copia los elementos de un arreglo no empacado en un arreglo empacado.

**Palabra**

En computación se refiere a una localidad de memoria o a un grupo de localidades adyacentes.

**Palabra reservada**

Identificador reservado para aplicaciones especiales en Pascal, como PROGRAM, VAR, BEGIN y END.

**Parámetro**

Valor o variable que aparece en la lista de parámetros de un encabezado de procedimiento o función, o en la lista de parámetros de una invocación de procedimiento o función.

**Parámetro de archivo**

Parámetro usado en la proposición PROGRAM para conectar una variable de archivo a un archivo externo.

**Parámetro de arreglo conformante**

Parámetro que se puede usar en los procedimientos y funciones del Pascal con el fin de permitir que distintas invocaciones tengan argumentos con límites diferentes.

**Parámetro de entrada (dónde valor)**

Parámetro asociado con los datos de entrada de un procedimiento o función.

**Parámetro de salida (de variable)**

Parámetro asociado a la salida de un procedimiento. Ocupa la misma memoria que el parámetro verdadero correspondiente.

**Parámetro de valor**

En Pascal un parámetro que representa datos de entrada para un procedimiento o función.

**Parámetro de variable**

En Pascal, un parámetro que representa datos de salida de un procedimiento o función. También puede representar datos de entrada para un procedimiento o función.

**Parámetro formal**

Identificador que aparece en la lista de parámetros de un encabezado de procedimiento o función.

**Parámetro verdadero**

Valor real que se pasa mediante la invocación a una función o procedimiento.

**Parte fija**

Parte de un registro variante común a todas las posibles formas de ese registro.

**Parte variante**

Parte de un registro variante que puede cambiar dependiendo del registro de que se trate.

**Pila**

Estructura de datos dinámica en la que los elementos se agregan o quitan en un extremo (parte superior de la pila). También se le llama primero que entra, último que sale.

**Postcondición**

Aserción que se cumple después de ejecutarse una secuencia de proposiciones.

**Postorden**

Se refiere al algoritmo de recorrido de árboles que visita primero el subárbol izquierdo, después el subárbol derecho y por último la raíz.

**Precondición**

Aserción que se cumple antes de ejecutarse una secuencia de proposiciones.

**Preorden**

Se refiere al algoritmo de recorrido de árboles que visita primero la raíz, después el subárbol izquierdo y finalmente el subárbol derecho.

**Procedimiento**

Subprograma con un nombre y parámetros opcionales que se puede llamar o invocar para que se ejecute.

**Procedimientos de biblioteca**

Programas de uso frecuente que residen en la biblioteca de un sistema.

**PROCEDURE**

Palabra reservada en Pascal que se usa para declarar procedimientos.

**Procesamiento por lote**

Modo de ejecutar programas que no requiere la interacción directa entre el usuario y la computadora.

**Programa de computadora**

Conjunto de instrucciones que debe realizar (ejecutar) una computadora.

**Programa ejecutable**

Programa en lenguaje de máquina que ejecuta directamente la computadora.

**Programa fuente**

Programa escrito en un lenguaje de computación antes de ser traducido a lenguaje de máquina.

**Programa objeto**

Programa en lenguaje de máquina que resulta de la traducción del programa fuente.

**Programa principal**

Proposiciones ejecutables de un programa en Pascal que aparecen después de todas las declaraciones de constantes, tipos, variables, procedimientos y funciones. El programa principal es siempre la parte del programa completo en la que se inicia la ejecución.

**Proposición CASE**

Proposición del Pascal que permite elegir una acción de entre varias posibles.

**Proposición de asignación**

Proposición del Pascal que incluye el operador de asignación : = y sustituye el contenido de la variable que está a la izquierda del operador por el valor de la expresión que está a la derecha del operador.

**Proposición ejecutable**

En un programa en Pascal, proposición que la computadora traduce a instrucciones en lenguaje de máquina, a diferencia de las declaraciones CONST, TYPE y VAR que no producen instrucciones en lenguaje de máquina.

**Proposición FOR**

Estructura de control cíclica o estructura de control definida en la que la variable de control del ciclo cambia automáticamente desde un valor inicial hasta un valor final.

**Proposición GOTO**

Proposición del Pascal que transfiere incondicionalmente el control a una proposición etiquetada.

**Proposición IF-THEN**

Proposición del Pascal que se usa para ejecutar condicionalmente una proposición si el valor Booleano especificado es true.

**Proposición IF-THEN-ELSE**

Proposición del Pascal que se usa para ejecutar de manera selectiva una de dos proposiciones depen-

diendo de un valor booleano especificado.

**Proposición REPEAT-UNTIL**

Estructura de control indefinida que se usa en los ciclos y es similar a la proposición WHILE.

**Proposición vacía**

Proposición que se indica mediante un signo de punto y coma y que no implica acción alguna.

**Proposición WHILE**

Estructura de control indefinida que se usa en ciclos escritos en Pascal.

**Proposición WITH**

Proposición del Pascal que permite referirse a los componentes de un registro de manera abreviada.

**Proposiciones IF anidadas**

Proposiciones IF contenidas dentro de otras proposiciones IF.

**Prueba ascendente**

Prueba de un programa que consiste en ejercitar los procedimientos de bajo nivel (los de la parte inferior del árbol de diseño) antes de unirlos a otros.

**Prueba descendente**

Prueba de un programa en la que se efectúan primero los procedimientos del nivel más alto del árbol de diseño.

**Puntero**

Tipo de datos del Pascal que contiene la dirección de memoria de una variable de un tipo especificado.

**Put**

Procedimiento estándar del Pascal que agrega el contenido de la variable de almacenamiento temporal al archivo correspondiente.

**Raíz**

Nodo de un árbol que indica la parte superior del mismo.

**Read**

Procedimiento estándar del Pascal que permite capturar datos de un archivo.

**ReadIn**

Procedimiento estándar del Pascal que se emplea exclusivamente con archivos de texto y es similar al procedimiento read, con la diferencia de que al terminar adelanta la ventana de archivo hasta más allá del siguiente fin de línea.

**Real**

Tipo de datos en Pascal que consta de números con punto decimal, fracción decimal y exponentes.

**Recorrido**

Se refiere a visitar los nodos de un árbol.

**Refinación por pasos**

Proceso de descomponer un problema en pasos mediante la refinación continua.

**Registro**

Tipo de datos estructurado del Pascal con un número fijo de componentes a los cuales se tiene acceso por su nombre. Los componentes pueden ser de tipos diferentes.

**Registro variante**

Registro que tiene dos partes distintas: una fija y una variante. Los componentes de un registro variante pueden cambiar de un registro a otro.

**Registros jerárquicos**

Registros que contienen otros registros como componentes (registros anidados).

**Reset**

Procedimiento estándar en Pascal que prepara un archivo para captura o lectura.

**Rewrite**

Procedimiento estándar en Pascal que prepara un archivo para salida o grabación.

**Secuencia de colación**

Ordenamiento de un determinado conjunto de caracteres (por ejemplo, ASCII o EBCDIC).

**Selección**

Estructura de control fundamental que selecciona una de varias alternativas dependiendo de una condición.

**Selector**

En una posición CASE, la variable que sirve para elegir el curso de acción.

**Semilla**

Valor inicial que se usa para generar números pseudoaleatorios.

**Seudocódigo**

Mezcla de español e instrucciones de cómputo que se usa para expresar un algoritmo.

**Sintaxis**

Reglas formales para construir proposiciones legales en un lenguaje de cómputo.

**Sistema interactivo**

Sistema de cómputo en el que el usuario interactúa directamente con el sistema operativo, por lo regular mediante una terminal de video.

**Sistema operativo**

Conjunto de programas que administra un sistema de cómputo y los programas del sistema.

**Software**

Conjunto de programas escritos para una computadora.

**Subárbol**

Parte de un árbol que también es un árbol.

**Subconjunto**

Si todos los elementos del conjunto A son elementos del conjunto B, el conjunto A es un subconjunto del conjunto B.

**Subíndice**

Valor o índice que se usa para tener acceso a los componentes de un arreglo.

**Texto**

Archivo que contiene caracteres y marcas de fin de línea, dividido en líneas, cada una de las cuales termina con una marca de fin de línea.

**Tipo base**

Tipo de los valores potenciales que puede asumir una variable de conjunto.

**Tipo de datos**

Representación de los datos (por ejemplo números y caracteres) como un tipo de objeto bien definido especificado mediante reglas de sintaxis.

**Tipo de datos de subescala**

Tipo de datos que consiste de una escala dada de valores de un tipo de datos ordinal.

**Tipo de datos huésped**

El tipo de datos ordinal del que se pueden derivar tipos de datos de subescala.

**Tipo definido por el usuario**

Véase tipo enumerado.

**Tipo enumerado**

Tipo de datos definido por el usuario y que consta de un conjunto ordenado de constantes nuevas.

**Tipos compatibles**

Dos tipos son compatibles si son el mismo tipo, o uno es una subescala del otro, o ambos son subescalas del mismo tipo huésped.

**Tipos de datos estructurados**

En Pascal uno de los tipos arreglo, registro, archivo o conjunto.

**Tipos de datos ordinales**

Tipos de datos que constan de un conjunto ordenado de valores distintos que tienen un primer y un último elemento.

**Tipos de datos simples**

En Pascal, cualquier tipo ordinal definido por el usuario o los tipos entero, real, booleano y de carácter.

**Tipos idénticos**

Se refiere a los objetos de programación que tienen el mismo nombre de tipo.

**Transferencia incondicional**

Transferencia incondicional de control a un punto determinado de un programa. En Pascal, esto se rea-

liza mediante la proposición GOTO.

**Transportabilidad**

Capacidad de transportar un programa de un sistema de cómputo a otro con un mínimo de modificaciones.

**Truncar**

Proceso de pasar por alto o eliminar la parte fraccionaria de un número real.

**Umbral**

Valor que se usa para determinar la terminación de un ciclo.

**Unidad aritmético-lógica (UAL)**

Componente de la UCP que lleva a cabo las operaciones de aritmética y lógica.

**Unidad central de procesamiento (UCP)**

Componente principal de la computadora que dirige y controla el procesamiento de la información que realiza la computadora.

**Unidad de control**

Componente de la UCP que dirige la ejecución de las instrucciones de cómputo.

**Unidad de memoria**

Componente importante de la computadora formada por muchas celadas de memoria en las que se almacenan programas de computadora y datos.

**Unión**

Operación sobre dos o más conjuntos que forma un conjunto que contiene todos los elementos de los conjuntos que son los operandos.

**Unpack**

Procedimiento estándar del Pascal que convierte un arreglo empacado de elementos en un arreglo no empacado.

**Variable**

Objeto de programa cuyo valor puede cambiar durante la ejecución.

**Variable booleana**

Variable de tipo booleano.

**Variable de almacenamiento temporal de archivo**

Variable que sirve para tener acceso al siguiente componente secuencial de un archivo antes de que lo procese realmente una posición read o write.

**Variable de archivo**

Variable de tipo FILE usada para tener acceso a un archivo en las proposiciones reset, rewrite, read o write.

**Variable de cadena**

Variable que se declara como arreglo empacado de caracteres.

**Variable de control de ciclo**

Variable empleada para controlar el número de iteraciones de un ciclo.

**Variable de puntero**

Variable de tipo puntero.

**Variable de referencia**

Variable a la que se tiene acceso a través de una variable de puntero.

**Variable dinámica**

Variable a la que se tiene acceso únicamente a través de una variable de puntero.

**Variable global**

Variable que se declara en un procedimiento o programa principal y por

tanto está accesible en cualquier procedimiento o función dentro del alcance de la variable.

**Variable local**

Variable que se declara en un procedimiento (o función) y que se conoce únicamente dentro de ese procedimiento (o función).

**Variante de ciclo**

Aserción que cambia después de cada iteración de un ciclo.

**Ventana de archivo**

Parte de un archivo a la que se puede tener acceso a través de las variables de almacenamiento temporal.

**Verificación de programas**

Rama de la computación que se ocupa de demostrar la calidad de correctos de los programas.

**Write**

Procedimiento estándar en Pascal que permite la salida de datos a un archivo.

**Writeln**

Proposición estándar en Pascal similar a write que también agrega un carácter de fin de línea a los archivos de texto como operación final.



# RESPUESTAS A LOS EJERCICIOS

## SECCIÓN 2.2

1.
  - a) válida
  - b) válida
  - c) no válida
  - d) no válida
  - e) no válida
  - f) no válida
  - g) no válida
  - h) válida
  - i) no válida
  - j) no válida
2.
  - a) debe comenzar con letra
  - b) no se puede usar \*, ya que es un operador aritmético
  - c) palabra reservada
  - d) debe comenzar con letra
  - e) identificador estándar
  - f) palabra reservada
  - g) función estándar
  - h) debe comenzar con letra
3.
  - a) válida
  - b) no válida
  - c) válida
  - d) no válida
4.
  - a) CONST días = 7;
  - b) CONST peso = 185;
  - c) CONST impventa = 0.08;
  - d) CONST alumnos = 50;
5.
  - a) válida
  - b) no válida
  - c) no válida
  - d) válida
6.
  - a) VAR nss : integer;
  - b) VAR año : integer;
  - c) VAR tasaimp : real;
  - d) VAR califprom : real;
7.
  - a) PROGRAM  
BEGIN  
END  
VAR  
CONST
  - b) *integer*  
*input*  
*output*  
*read*  
*writeln*

## SECCIÓN 2.3

1. a) válida  
 b) no válida  
 c) no válida  
 d) válida siempre que  $-55555$  sea  
 $\geq -\text{máxint}$   
 e) no válida  
 f) no válida  
 g) válida  
 h) no válida  
 i) válida
2. a) válida  
 b) no válida  
 c) válida  
 d) no válida  
 e) válida  
 f) no válida  
 g) válida  
 h) no válida  
 i) no válida
3. a) no válida (faltan comillas)  
 b) válida  
 c) válida (constante de cadena de caracteres)  
 d) válida  
 e) válida  
 f) válida (constante de cadena de caracteres)
4. a) 6210 o  $6.21e3$   
 b) 0.0015 o  $0.15e - 2$   
 c) 1660000.0 o  $1.66e6$   
 d) 0.000000224 o  $22.4 e - 8$
5. a)  $2.6 \times 10^3 = 2600$   
 b)  $-12 \times 10^{-2} = -0.12$   
 c)  $4.56 \times 10^{10} = 45600000000.0$   
 d)  $-66.5 \times 10 = -665.0$
6. a) necesita un dígito a la derecha del punto decimal  
 b) necesita un dígito a la izquierda del punto decimal  
 c) el exponente no es entero  
 d) el signo de menos debe ir antes del número  
 e) no hay dígito después del punto decimal  
 f) . . . no puede seguir a un número
7. a) válida, char  
 b) no válida  
 c) válida, boolean  
 d) válida, integer  
 e) no válida  
 f) válida, integer  
 g) válida, real  
 h) válida, char  
 i) no válida

## SECCIÓN 2.4

1. a) válida  
 b) no válida  
 c) válida  
 d) válida  
 e) no válida  
 f) válida  
 g) no válida
2. a) 2  
 b) 3  
 c) 5  
 d) 8  
 e) 3  
 f) 1
3. a) válida  
 b) no válida
4. a) 4  
 b) 11  
 c)  $-4$   
 d) 10  
 e)  $-13$   
 f) 25  
 g) 12  
 h)  $-1$
5. a) 3.0  
 b) 2.5

6. a)  $(x + y)/(y/z + 3)$   
 b)  $(-b - \text{sqrt}(\text{sqr}(b) - 4 * a * c))/(2 * a)$   
 c)  $\text{sqr}(\text{sqr}(x) + \text{sqr}(y))$   
 d)  $(x + y - z)/(2 * x + \text{sqr}(y) - x * y * z)$

7. (a)  $\frac{abc}{de}$   
 (b)  $x + \frac{y}{z}(a+b)$   
 (c)  $b^2 - 4ac$   
 (d)  $a - \frac{bc}{d} + ef$

8. a)  $(2 * b) - a + ((b * d)/e)$   
 b)  $f - ((e * d)/c) + (b * a)$   
 c)  $(a * a) - ((b * c)/d) + \text{sqrt}(e + (2 * f))$   
 d)  $(d * (a + b)) - \text{trunc}((e * f)/g) + (b/(a - (b * c)))$   
 e)  $((6 \text{ DIV } 2) * (5 \text{ MOD } 3)) - (2 * 3)$   
 f)  $((2 * 3) \text{ MOD } ((8 \text{ DIV } 3) + 1)) + \text{round}(6.5)$

9. a) válida  
 b) no válida  
 c) no válida  
 d) válida  
 e) válida  
 f) no válida  
 g) válida  
 h) no válida  
 i) no válida

10. a) 11  
 b) 6  
 c) 36  
 d) 2

## SECCIÓN 3.1

1. x = 8  
 y = 7  
 z = 2  
 a = 1  
 b = 1  
 c = 4  
 d = 3
2. a) no hay errores  
 b) se presentará un error cuando se asigne 2.3 a b  
 c) se presentará un error cuando se asigne 2.1 a c  
 d) se presentará un error porque no quedarán datos de entrada para asignar a z

3. Valor1 es 3  
 valor2 es 5  
 La suma es 8
4. 86 39  
 a =  
 3.250000000000e + 01  
 a = 86 b = 3.250000000000e + 01
5. Los valores son 0 2 15.200000000000e + 00  
 | línea 1, columna 1  
 3.600000000000e + 00 4.100000000000e + 00  
 | línea 1, columna 62  
 La suma es 1.290000000000e + 01  
 | línea 3, columna 1  
 El producto es 0  
 | línea 3, columna 1
6. x = 12
7. a = 2 b = 15  
 c = 1  
 d = 8
8. Valor 4 = 18
9. e
10. a
11. x = 12  
 y = 7  
 z = 11
12. HOLA  
 R = 6.10  
 S = 7.2

## SECCIÓN 4.2

1. La velocidad es 50mph.  
 La velocidad es 40mph.
2. 2 4 2  
 4 4 2
3. PROCEDURE trisuma;  
 BEGIN  
 total := num1 + num2 + num3  
 END
4. PROCEDURE prom;  
 BEGIN  
 promedio := total DIV 3  
 END

5. PROGRAM calcular (input, output);  
 VAR  
     núm1, núm2, núm3 : integer;  
     total : integer;  
     promedio : integer;  
 PROCEDURE trisuma;  
 BEGIN  
     total := núm1 + núm2 + núm3  
 END;  
 PROCEDURE prom;  
 BEGIN  
     promedio := total DIV 3  
 END;  
 BEGIN (\* principal \*)  
     writeln ('Por favor escriba tres números enteros');  
     readln (núm1, núm2, núm3);  
     trisuma;  
     prom;  
     writeln ('El valor promedio es', promedio)  
 END.

Ejecución:

```
Por favor escriba tres números enteros
34 67 159
El valor promedio es      86
```

6. PROCEDURE círculo;  
 BEGIN  
     área := pi \* sqr(r)  
 END
7. El procedimiento valor determina la cantidad de dinero que se tiene, dependiendo del número de monedas de cinco centavos (níqueles).
8. PROCEDURE dinero;  
 BEGIN  
     dólares := (0.05 \* níqueles) + (0.1 \* dieces) + (0.25 \* cuartos)  
 END

## SECCIÓN 4.3

- x, y y z son los parámetros de entrada (de valor).  
 a, b y c son los parámetros de salida (variables).
- Los parámetros formales son los que se listan en el encabezado del procedimiento. Funcionan como representantes de los valores verdaderos.  
 Los parámetros verdaderos son los valores reales que se pasan al procedimiento cuando se ejecuta.
- Los parámetros formales son tiempo, espacio, día y signo. Los parámetros verdaderos son 3.5, 6.0, hora y 'Z'.

```
tiempo = 3.5
espacio = 6.0
día = hora
signo = 'Z'
```

4. Al invocarse prueba, pasa el valor 2.0 a y. El problema es que 2.0 es una constante real y y es un parámetro entero.
  5. No es posible pasar 4.0 al parámetro variable z. Debe sustituirse 4.0 por el nombre de una variable real.
  6. La constante de caracteres Z debe estar encerrada entre apóstrofos ('Z').
  7. tum, num y temp.
  8. reloj
  9. a, b y c  
a y b son parámetros de entrada  
c es parámetro de salida
  10. tum = 1, num = 2, temp
  11. num = 1, tum = 0, temp
  12. 

1	2	4	3
4			
1	0	2	1
2			

Se produjeron cuatro líneas de salida.

## **SECCIÓN 4.4**

- |    |   |   |   |
|----|---|---|---|
| 1. | 1 | 2 | 3 |
|    | 3 | 5 | 4 |

  2. PROCEDURE cubo (largo : real; VAR volumen : real);  
BEGIN  
    volumen := largo \* largo \* largo  
END
  3. PROCEDURE magnitud (x1, x2 : real; VAR distancia : real);  
BEGIN  
    magnitud := abs (x1 - x2)  
END
  4. PROCEDURE dígito (número : integer; VAR másbajo : integer);  
BEGIN  
    másbajo := número MOD 10  
END
  5. PROCEDURE círculo (radio : real; VAR área : real);  
BEGIN  
    área := 3.14 \* sqr (radio)  
END
  6. PROCEDURE rectángulo (largo, ancho ; real; VAR área : real);  
BEGIN  
    área := largo \* ancho  
END

## 7. PROGRAM. descubierto (input, output);

VAR

```
piso, tapete1, tapete2: real;
descub : real;
```

PROCEDURE círculo (radio : real; VAR área : real);

BEGIN

```
área := 3.14 * sqr (radio)
```

END;

PROCEDURE rectángulo (largo, ancho : real; VAR área : real);

BEGIN

```
área := largo * ancho
```

END

BEGIN (\* principal \*)

```
círculo (1.0, tapete1);
```

```
círculo (2.0, tapete2);
```

```
rectángulo (12.0, 15.0, piso);
```

```
descub := piso - tapete1 - tapete2;
```

```
writeln ('El área descubierta es ', descub)
```

END.

Ejecución:

El área descubierta es 1.64300e + 02

**SECCIÓN 5.1**

- |                 |  |
|-----------------|--|
| 1. a) verdadera | 4. a) verdadera                        |
| b) verdadera    | b) verdadera                           |
| c) falsa        | c) falsa                               |
| d) verdadera    | d) verdadera                           |
| e) falsa        | 5. a) (NOT indic) OR (NOT conmuta)     |
| f) falsa        | b) indic OR (prueba AND (NOT conmuta)) |
| 2. a) verdadera | c) NOT ((indic AND conmuta) OR prueba) |
| b) verdadera    | d) prueba OR (conmuta AND indic)       |
| c) verdadera    | 6. a) NOT (1 < num)                    |
| d) falsa        | b) (2 > num) OR (num < - 2)            |
| 3. a) falsa     | c) (NOT (0 < num)) OR (num > 10)       |
| b) falsa        | d) (1 > num) AND (num < 0)             |
| c) falsa        |  |
| d) verdadera    |  |
| e) falsa        |  |

**SECCIÓN 5.2**

- |   |                                  |
|---|----------------------------------|
| a) Extrema  | 4. a) falsa                      |
| b) Extrema  | b) falsa                         |
| c) Media  | c) falsa                         |
| 2. IF (a > b) AND (a > c)<br>THEN write ('A es el mayor') | d) verdadera                     |
| 3. x = 7  | 5. Se exhibe 'Nunca llega aquí.' |
| y = 9   | 6. Nada se exhibe.               |
|   | 7. B                             |

## SECCIÓN 5.3

1. El valor es 5.
2. CASE k OF
 

```
0 : r : = r + 1;
1 : s : = s + 1;
2,3,4 : t : = t + 2
END
```

3. b)
4. CASE calif OF
 

```
D, F : writeln ('Muy mal.');
C, B : writeln ('Buen trabajo.');
A   : writeln ('Excelente trabajo.')
END
```

## SECCIÓN 6.1

1. - 2
2. 9        15
3. 17         $\sqrt{25}$
4. 30        2        0
5. 4        3
6. Suma = 18, valor = 0

## SECCIÓN 6.2

1. Una vez
2. Primitiva
3. -1
4. c)
5. 1        3        2        5        3
6. 4        4
 

3	4
---	---

## SECCIÓN 7.1

1. c)
2. d)
3. b)
4. a)
5. 4
 

8
---
6. 4        4        2        4
 

1	0	4
---	---	---

11	11	6	6
6	0	11	
7. b)			
8. b)			
9. d)			
10. 15	20	4	30
2	15	3	30
8	10	5	6
6	8	3	30

## SECCIÓN 7.2

- |              |                   |                        |
|--------------|-------------------|------------------------|
| 1. a) falsa  | 2. 2              | 5. d)                  |
| b) falsa     | 4                 |                        |
| c) verdadera | 3. La cuenta es 3 | 6. 2        3        4 |
| d) falsa     |                   |                        |
| e) falsa     | 4. c)             |                        |

7. FUNCTION digfinal (x : integer) : integer;  
 BEGIN  
     digfinal := x MOD 10  
 END

8. PROGRAM notas (input, output);  
 VAR  
 i, calif, másbajo : integer;  
 BEGIN  
     i := 0;  
     másbajo := 101;  
     WHILE NOT eof (input) DO  
     BEGIN  
         readln (calif);  
         IF (calif >= 0) AND (calif <= 100)  
         THEN BEGIN  
             i := i + 1;  
             IF calif < másbajo  
                 THEN másbajo := calif  
         END  
         ELSE writeln ('Error, la calificación es demasiado alta.')  
     END;  
     writeln ('Número total de calificaciones = ', i);  
     writeln ('La calificación mínima es ', másbajo)  
 END.

## SECCIÓN 7.3

1. 15

2. 50

3. a)  $1! = 1$

$n! = n * (n - 1)!$  para  $n > 1$

b) PROGRAM nfact (input, output);  
 VAR

    x, m : integer;

FUNCTION fact (n : integer) : integer;  
 BEGIN

    IF n <= 1

    THEN fact := 1

    ELSE fact := fact (n - 1) \* n

END;

BEGIN

    readln (m);

    x := fact (m);

    writeln (m, '! = ', x)

END.

4. a) FUNCTION potencia (x : real, n : integer) : real;

(\* Función no recursiva para calcular la N potencia de x \*)

## VAR

i : integer;  
resp : real;

## BEGIN

resp := 1;  
FOR i := 1 TO n DO resp := resp \* x;  
potencia := resp

## END

b)  $x^1 = x$

$x^n = x \cdot x^{n-1}$  para  $n > 1$

c) FUNCTION potencia (x : real; n : integer) : real;

(\* función recursiva para calcular la enésima potencia de x \*)

## BEGIN

IF n = 1

THEN potencia := x

ELSE potencia := x \* potencia (x, n - 1)

## END

5. FUNCTION fib (n: integer) : integer;

(\* calcular el enésimo número de Fibonacci \*)

## BEGIN

IF (n = 1) OR (n = 2)

THEN fib := 1

ELSE fib := fib (n - 1) + fib (n - 2)

## END

## SECCIÓN 8.1

1. a) TYPE estadoa = (Alabama, Alaska, Arizona, Arkansas)  
b) TYPE pariente = (Primo, Papá, Mamá, Abuela, Abuelo)  
c) TYPE bebida = (Pepsi, Coca, SevenUp, Sidral)  
d) TYPE sabor = (Chocolate, Vainilla, Fresa)
2. a) no válida  
b) no válida  
c) no válida  
d) válida  
e) no válida  
f) válida
3. a) no válida  
b) válida  
c) no válida  
d) válida  
e) no válida  
f) válida
4. a) no válida  
b) no válida  
c) válida  
d) no válida  
e) válida

6. Viaje a Chicago

No se recomienda viajar

7. TYPE mes (Ene, Feb, Mar, Abr, May, Jun, Jul, Ago, Sep, Oct, Nov, Dic);  
PROCEDURE exhibir (nombre : mes);  
BEGIN

CASE nombre OF

Ene : writeln ('Enero');  
 Feb : writeln ('Febrero');  
 Mar : writeln ('Marzo');  
 Abr : writeln ('Abril');  
 May : writeln ('Mayo');  
 Jun : writeln ('Junio');  
 Jul : writeln ('Julio');  
 Ago : writeln ('Agosto');  
 Sep : writeln ('Septiembre');  
 Oct : writeln ('Octubre');  
 Nov : writeln ('Noviembre');  
 Dic : writeln ('Diciembre');

END

END

**SECCIÓN 8.2**

1. a) noneg : 0..máxint;

b) grande : 101..máxint;

c) mitad1 : 'A'..'M';

d) mitad2 : 'M'..'Z';

2. {99, 100, 101, ...999, 1000}

3. a) sí

b) no

4. la escala de letra es 'A', 'B', 'C', 'D', ..., 'Z'

la escala de negativo es de - máxint hasta - 1

la escala de unodsígitos es '0', '1', '2', '3'

la escala de temp es de - 32 a 32

la escala de primavera es Mar, Abr, May, Jun

la escala de semestre es Ene, Feb, Mar, Abr, May, Jun

5. d)

6. e)

7. PROGRAM palindroma (input, output);

TYPE

dígitos = '0'..'9';

VAR

car1, car2, car3, car4, car5 : dígitos;

BEGIN

readln (car1, car2, car3, car4, car5);

IF (car1 = car5) AND (car2 = car4)

```

THEN writeln ('La secuencia es un palindroma numérico')
ELSE writeln ('La secuencia no es un palindroma numérico')
END.

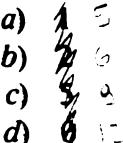
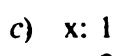
```

## SECCIÓN 8.3

1. a) 7  
b) 2  
c) - 2  
d) '7'  
e) no definido  
f) no definido
2. a) - 2  
b) 100  
c) '0'  
d) '4'  
e) 6  
f) 'B'
3. a) 0  
b) 4  
c) cudol  
d) no definido  
e) no definido  
f) 5
4. a) 7  
b) 0  
c) '3'  
d) '0'
5. 2001  
37
6. Se ejecutará con error
7. Se ejecutará sin error
8. PROCEDURE convertir (VAR núm : integer);
 VAR
 uncar : char;
 BEGIN
 núm := 0;
 uncar := ' ';
 WHILE (uncar < '0') OR (uncar > '9') DO read (uncard);
 WHILE (uncar >= '0') AND (uncar <= '9') DO
 BEGIN
 núm := núm \* 10 + ord(uncar) - ord('0');
 read (uncar)
 END
 END

1.
  - a) 0, 1, 2, 3
  - b) 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 13, 15
  - c) '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'
  - d) 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M'
  - e) '0', '1', '2', '3', '4', '5', '9'
2.
  - a) no válida
  - b) válida
  - c) no válida
  - d) válida
  - e) no válida
3.
  - a) falsa
  - b) verdadera
  - c) verdadera
  - d) falsa
4.
  - a) falsa
  - b) verdadera
  - c) falsa
  - d) verdadera
5.
  - a) falsa
  - b) falsa
  - c) verdadera
  - d) ~~falsa~~  $\top$
6. 2  
2
7. PROGRAM nodígito (input, output);  
VAR  
 contador : integer;  
 i : integer;  
 car : char;  
BEGIN  
 i := 0;  
 contador := 0;  
 WHILE (NOT eoln (input)) AND (i < 80) DO  
 BEGIN  
 read (car);  
 i := i + 1;  
 IF NOT (car IN ['0'..'9'])  
 THEN contador := contador + 1 (\* no es digito \*)  
 END;  
 writeln ('El número total de caracteres no dígitos es ', contador);  
END.

## SECCIÓN 9.1

1. arreglox tiene 100  
arregloy tiene 7  
arregloz tiene 5
2. Se leen nueve números y se introducen en el arreglo núm.
3. a) válida  
b) válida  
c) no válida  
d) válida si datos[0] está en ['A'..'Z']  
e) válida  
f) no válida  
g) válida si lista1['A'] está en [-3..3]
4. a)   
b)   
c)   
d) 
5. d)
6. c) x: 1 6 10 13  
y: 2 7 11 14
7. c)
8. TYPE fra = ARRAY [1..100] OF char;  
 FUNCTION alfa (frase : fra; largo : integer; letra : char) : integer;  
 (\* Determinar cuántas veces aparece letra en frase \*)  
 VAR  
     contador, i : integer;  
 BEGIN  
     contador := 0;  
     FOR i : 1 TO largo DO  
         IF frase[i] = letra  
             THEN contador := contador + 1;  
     alfa := contador  
 END
 (\* recorrer el arreglo \*)  
    (\* ¿se encontró una? \*)  
        (\* sí \*)  
    (\* entregar resultado final \*)

## SECCIÓN 9.2

1. a) Se requerirán seis comparaciones para localizar el valor 12 mediante una búsqueda lineal.  
b) Se requerirán tres comparaciones para localizar el valor 12 mediante una búsqueda binaria después de clasificar los valores.
2. La eficiencia es la misma.
3. PROGRAM fusión (input, output);  
 CONST  
     tamarr = 100; (\* tamaño máximo combinado \*)  
 TYPE  
     arreglote = ARRAY [1..tamarr] OF integer;

VAR

```
arr1, arr2 : arreglote;
largo : arreglote;
i : integer;
inicio : integer;
m, n, total : integer;
```

PROCEDURE clasif (tamarr : integer; VAR tabla : arreglote);

VAR

temp, j, mayor : integer;

FUNCTION hallamáx (último : integer; VAR tabla : arreglote) : integer;

VAR

indmáx, índice : 1..tamarr;

BEGIN

indmáx := 1;

FOR índice := 2 TO último DO

IF tabla[índice] &gt; tabla[indmáx]

THEN indmáx := índice;

hallamáx := indmáx

END;

BEGIN

FOR j := tamarr DOWNTO 2 DO

BEGIN

mayor := hallamáx (j, tabla);

temp := tabla[mayor];

tabla[mayor] := tabla[j];

tabla[j] := temp

END

END;

PROCEDURE copiar (inicio, final : integer; tabla : arreglote; VAR largom : arreglote);

(\* este procedimiento agrega un arreglo al otro \*)

VAR

i, j : integer;

BEGIN

FOR i := 1 TO final DO

BEGIN

largom[inicio] := tabla[i];

inicio := inicio + 1

END

END;

BEGIN

(\* programa principal \*)

writeln ('Por favor especifique los tamaños de los arreglos.');

readln (m, n);

total := m + n;

IF total &gt; tamarr

THEN writeln ('Los arreglos son demasiado grandes.')

ELSE BEGIN

writeln ('Escriba los elementos del primer arreglo.');

FOR i := 1 TO m DO read (arr1[i]);

writeln ('Escriba los elementos del segundo arreglo.');

FOR i := 1 TO n DO read (arr2[i]);

readln;

```

    clasif (m, arr1);
    clasif (n, arr2);
    inicio := 1;
    copiar (inicio, m, arr1, largo);
    copiar (m + 1, n, arr2, largo);
    clasif (m + n, largo);
    FOR i := TO m + n DO writeln (largo[i])
  END
END.

```

4. FUNCTION hallamín (último : integer; VAR tabla : lista) : integer;
(\* hallar el índice del elemento más pequeño de tabla[1..último] \*)
 VAR
 indín, índice : 1..tamarr;
 BEGIN
 indmín := 1;
 FOR índice := 2 TO último DO
 IF tabla[índice] < tabla[indmín]
 THEN indmín := índice;
 hallamín := indmín
 END
5. PROCEDURE clasif (tamarr : integer; VAR tabla : lista);
(\* clasificación por selección de los elementos del arreglo tabla[1..tamarr] \*)
 VAR
 temp, j, menor : integer;
 BEGIN
 FOR j := tamarr DOWNTO 2 DO
 BEGIN
 mejor := hallamín (j, tabla);
 temp := tabla[menor];
 tabla[menor] := tabla[j];
 tabla[j] := temp
 END
 END

## SECCIÓN 9.3

1. matriz contiene 121 elementos.
2. a)
3. 5  
5  
5
4. a) Existen 90 componentes.  
b) no válida  
c) no válida  
d) no válida: índice fuera de escala  
e) no válida
5. TYPE matriz = ARRAY [1..10, 1..10] OF integer;  
PROCEDURE cambio (m, n : integer; VAR matrizr : matriz);

(\* intercambiar filas m y n de matrizr \*)

```

VAR
    temp : matriz;
    i, j : integer;
BEGIN
    FOR i := 1 to 10 DO
        BEGIN
            temp[m, i] := matrizr[m, i];
            matrizr[m, i] := matrizr[n, i];
            matrizr[n, i] := temp[m, i]
        END
    END

```

6. PROGRAM trueque (input, output);  
 TYPE tridim = ARRAY [1..10, 1..20, 1..30] OF Boolean;

```

VAR
    índice1, índice2, índice3 : integer;
    lógica : tridim;
    i, j, k : integer;
    total, datos : integer;
    bien : Boolean;
PROCEDURE flipflop (arreglo : tridim);
(* invertir el sentido lógico de los elementos de arreglo *)
VAR
    i, j, k : integer;
BEGIN
    FOR i := 1 TO índice1 DO
        FOR j := 1 TO índice2 DO
            FOR k := 1 TO índice3 DO
                arreglo[i, j, k] := NOT arreglo[i, j, k]
END;
BEGIN (* principal *)
    writeln ('Especifique por favor las dimensiones del arreglo.');
    bien := true;
    readln (índice1, índice2, índice3);
    IF (índice1) > 10)
    THEN BEGIN
        writeln ('La primera dimensión del arreglo es demasiado grande.');
        bien := falsa
    END;
    IF (índice2) > 20)
    THEN BEGIN
        writeln ('La segunda dimensión del arreglo es demasiado grande.');
        bien := falsa
    END;
    IF (índice3) > 30)
    THEN BEGIN
        writeln ('La tercera dimensión del arreglo es demasiado grande.');
        bien := falsa
    END;
    IF bien
    THEN BEGIN

```

```
writeln ('Escriba los datos: 0 = falso, < > 0 = verdadero.');
FOR i := 1 TO índice1 DO
  FOR j := 1 TO índice2 DO
    FOR k := 1 TO índice3 DO
      BEGIN
        read (datos);
        lógica[i, j, k] := datos < > 0
      END;
      flipflop (lógica)
    END
  END.
```

## SECCIÓN 10.1

1. a) válida  
b) no válida  
c) válida  
d) válida  
e) válida
2. a)
3. TYPE palabra = PACKED ARRAY [1..20] OF char;
 FUNCTION comparar (VAR palabra1, palabra2 : palabra) : integer;
 BEGIN
 IF palabra1 < palabra2
 THEN comparar := -1
 ELSE IF palabra1 = palabra2
 THEN comparar := 0
 ELSE comparar := 1
 END
4. La función busca el primer espacio en blanco en el arreglo empacado comenzando por el final. El resultado es cero si no había espacios en todo el arreglo, o el índice del espacio si encontró uno.
5. pascal

## SECCIÓN 10.2

1. a)  $2 + 2 = 4$   
b) La hora ha llegado  
c) Ave. Bolívar 2500  
d) Nueva York, Nueva York  
e) blancoblanco
2. TYPE frase = PACKED ARRAY [1..80] OF char;
 FUNCTION ctablancos (VAR enunciado : frase) : integer;
 VAR
 cuenta, índice : integer;
 BEGIN
 cuenta := 0;

```

FOR índice := 1 TO 80 DO
  IF enunciado[índice] = ' '
    THEN cuenta := cuenta + 1;
ctablancos := cuenta
END

```

3. d)  
4. Salida:

```

PÉREZ
PÉREZPÉREZ
PÉREZ

```

```

5. PROCEDURE extapelido (VAR tabla : lista; tamarr : integer);
(* Extraer y exhibir el apellido de cada uno de los      *)
(* nombres completos de tabla[1] hasta tabla[tamarr], *) 
(* donde el tamaño de la lista es un parámetro.      *)
CONST
  blancos = '           '; (* 15 espacios *)
TYPE
  palabra = PACKED ARRAY [1..15] OF char;
VAR
  apellido ; palabra;          (* un apellido *)
  uncar : char;                (* un carácter de un nombre *)
  j,                      (* subíndice del nombre completo que se procesa *)
  índice : integer;            (* subíndice del primer nombre *)
  ind : integer;               (* subíndice del último nombre *)
BEGIN
  FOR j := 1 TO tamarr DO      (* para cada nombre *)
    BEGIN
      apellido := blancos;
      índice := 0;
      uncar := tabla[j, 1];
      WHILE (uncar < > ' ') AND (índice < 15) DO
        BEGIN
          índice := índice + 1;
          uncar := tabla[j, índice + 1]
        END;
      ind := 1;
      WHILE (uncar < > ' ') AND (ind < 15) DO
        BEGIN
          índice := índice + 1;
          apellido[ind] := tabla[j, índice + 1];
          ind := ind + 1
        END;
      writeln (apellido)
    END
  END
END

```

## SECCIÓN 11.1

1. a) TYPE

```

banco = RECORD
    nomdirec : PACKED ARRAY [1..100] OF char;
    númss : PACKED ARRAY [1..11] OF char;
    saldo : real;
    intacum : real
END

b) TYPE
dirtel = RECORD
    nombre : PACKED ARRAY [1..40] OF char;
    dir : PACKED ARRAY [1..50] OF char;
    núm : PACKED ARRAY [1..12] OF char;
END

c) TYPE
exprim = RECORD
    nomreal : PACKED ARRAY [1..40] OF char;
    alias : PACKED ARRAY [1..40] OF char;
    edad : integer;
    altura : integer;
    peso : integer;
    ojos : PACKED ARRAY [1..10] OF char;
    arrestos : integer
END

2. a) no válida, falta proposición END
b) válida
c) no válida, prueba aparece dos veces
d) válida

3. a) válida
b) no válida
c) no válida
d) no válida
e) no válida
f) válida
g) no válida

4. a) no válida
b) válida
c) válida
d) válida
e) no válida

5. TYPE
info = RECORD
    nombre : PACKED ARRAY [1..30] OF char;
    edad : 1..99;
    fechanac : PACKED ARRAY [1..10] OF char
END

VAR
    persona : info;
PROCEDURE lectura (VAR humano : info);
VAR
    n : PACKED ARRAY [1..30] OF char;
    d : PACKED ARRAY [1..10] OF char;

```

```
i : integer;
BEGIN
    i := 1;
    WHILE (i <= 30) DO
        BEGIN
            read (n[i]);
            humano.nombre[i] := n[i];
            i := i + 1
        END;
        read (humano.edad);
        i := 1;
        WHILE (i <= 10) DO
            BEGIN
                read (d[i]);
                humano.fechanac[i] := d[i];
                i := i + 1
            END
    END
END
```

## 6. TYPE

```
info = RECORD
    nombre : PACKED ARRAY [1..30] OF char;
    edad : 1..99;
    fechanac : PACKED ARRAY [1..10] OF char
END;

VAR
    persona : info;
PROCEDURE exhibir (VAR humano : info);
BEGIN
    writeln ('Nombre':35, 'Edad':7, 'Fecha Nacim.':15);
    writeln (humano.nombre:35, humano.edad:7, humano.fechanac:15)
END
```

## SECCIÓN 11.2

1. *a)* no válida  
*b)* válida  
*c)* válida  
*d)* no válida  
*e)* no válida  
*f)* válida  
*g)* no válida  
*h)* válida
  
2. *a)* válida  
*b)* no válida  
*c)* válida  
*d)* válida  
*e)* no válida  
*f)* no válida  
*g)* no válida  
*h)* válida

- i) no válida  
j) no válida
3. a) válida  
b) no válida, faltan caracteres  
c) no válida  
d) no válida, grupo necesita un índice  
e) válida  
f) válida
4. No es válida. CASE estado OF debe ser  
CASE grupo[índice].estado OF
5. Exhibirá 100 líneas, cada una de las cuales contiene 'T. J. Booker'
6. La primera línea tendrá 'T. J. Booker'.  
Las líneas de la 2 a la 100 tendrán 'Humpty Dumpty'.

## SECCIÓN 12.1

1. a) válida  
b) no válida, file es palabra reservada  
c) no válida  
d) válida  
e) válida  
f) no válida
2. a) válida  
b) no válida, falta ↑ antes del índice  
c) no válida, no se pueden asignar enteros a reales  
d) no válida, datcient debe tener subíndice  
e) válida
3. 1. es preciso ejecutar reset(archent) antes de read (archent, . . .)  
2. Eof(archent) y get(archent), no archsal  
3. declarar archent, archsal
4. TYPE n = FILE OF real;  
 VAR  
     núm : n;  
     númpos : n;  
 PROCEDURE duplicar (VAR número, positivo = n);  
 BEGIN  
     reset (número);  
     rewrite (positivo);  
     WHILE NOT eof(número) DO  
         BEGIN  
             IF número ↑ > 0  
                 THEN BEGIN  
                     positivo ↑ := número ↑;  
                     put (positivo)  
                 END;  
                 get (número);  
         END  
     END

## 5. TYPE

```
r = FILE OF real;
i = FILE OF integer;
```

## VAR

```
númreal : r;
númert : i;
```

```
PROCEDURE convertir (VAR nr : r; VAR ent : i);
```

## BEGIN

```
    reset (nr);
    rewrite (ent);
    WHILE NOT eof(nr) DO
        BEGIN
            ent := round (nr);
            get (nr);
            put (ent)
        END
    END
```

6. a) nomarch debe ser parámetro VAR.  
 b) no es posible asignar variables de archivo.  
 c) no es posible asignar enteros directamente a variables de caracteres.  
 d) Se debe usar  
     reset (nomarch); get(nomarch)  
     o bien  
     rewrite (nomarch); put(nomarch)  
 e) Se debe usar  
     reset (archnúm); get(archnúm)  
     o bien  
     rewrite (archnúm); put(archnúm)

## SECCIÓN 12.2

1. 4

2. La variable incóg cuenta el número de marcas de fin de línea (o líneas) del archivo de texto. Se presentará un error dado que la función eoln sólo se usa con archivos de texto.
3. PROCEDURE leetexto (VAR archent, archsal : texto);  
 (\* Leetexto lee datos de un archivo de texto el cual se copia a un \*)  
 (\* segundo archivo de texto. El segundo archivo tendrá nombres de \*)  
 (\* exactamente 72 caracteres, llenados con espacios si es preciso \*)

## VAR

```
car : ARRAY [1..72] OF char;
i, cuenta : integer;
```

## BEGIN

```
    reset (archent);
    rewrite (archsal);
    WHILE NOT eof (archent) DO
        BEGIN
            cuenta := 0;
            WHILE NOT eoln (archent) AND (cuenta < 72) DO
```

```

        BEGIN
            cuenta := cuenta + 1;
            read (archent, car[cuenta])
        END;
        readln (archent);
        FOR i := 1 TO cuenta DO write (archsal, car[i]);
        FOR i := cuenta + 1 TO 72 DO write (archsal, ' ');
        writeln (archsal)
    END
END

```

4. PROCEDURE noespacios (VAR archent : texto);  
(\* Leer un archivo de texto y exhibir el número de caracteres \*)  
(\* en cada línea que no son espacios en blanco. \*)

```

VAR
    cuenta : integer;
    car : char;
BEGIN
    reset (archent);
    WHILE NOT eof (archent) DO
        BEGIN
            cuenta := 0;
            WHILE NOT eoln (archent) DO
                BEGIN
                    read (archent, car);
                    IF car < > ' ' THEN cuenta := cuenta + 1
                END;
                writeln (cuenta);
                readln (archent)
            END
        END
    END

```

5. PROCEDURE rellenar (VAR archent : texto);  
(\* Rellenar lee archent y lo almacena en un arreglo empacado. \*)  
(\* En seguida rellena el resto del arreglo con espacios. \*)

```

VAR
    i : integer;
    car : PACKED ARRAY [1..72] OF char;
    datos : char;
BEGIN
    reset (archent);
    WHILE NOT eof (archent) DO
        BEGIN
            i := 0;
            WHILE NOT eoln (archent) DO
                BEGIN
                    i := i + 1;
                    read (archent, datos);
                    car[i] := datos
                END;
            WHILE i < 72 DO
                BEGIN
                    i := i + 1;

```

```

    car[i] := ' '
END;
readln (archent)
END

```

## SECCIÓN 13.1

1. a) válida  
b) válida  
c) no válida  
d) válida  
e) no válida
2. a) [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]  
b) [ ]  
c) [1, 2, 3, 4, 5, 7, 9, 11]  
d) [2, 4]  
e) [1, 3, 5, 7, 9, 11]  
f) [1, 2, 3, 4, 5, 7, 9, 11]  
g) [1, 3, 5, 7]  
h) [1, 3, 5, 7]  
i) [ ]  
j) [1, 2, 3, 4, 5, 7]  
k) [1, 3, 5, 7]
3. a) verdadera  
b) verdadera  
c) verdadera  
d) falsa  
e) falsa
4. a) [0, 1, 2, 10]  
b) [4, 5, 6]  
c) falsa  
d) falsa  
e) verdadera
5. [ ]  
[John]  
[Paul]  
[George]  
[Ringo]  
[John,Paul]  
[John,George]  
[John,Ringo]  
[Paul,George]  
[Paul,Ringo]  
[George,Ringo]  
[John,Paul,George]  
[John,Paul,Ringo]  
[John,George,Ringo]  
[Paul,George,Ringo]  
[John,Paul,George,Ringo]

## 6. PROGRAM español (input, output);

## TYPE

alfabeto = SET OF 'A'..'Z';

letras = 'A'..'Z';

## VAR

conja, conjb, conjc : alfabeto;

escala : letras;

car : char;

PROCEDURE simétrica (VAR conja, conjb, conjc : alfabeto);

(\* calcular la diferencia simétrica de conja y conjb \*)

## BEGIN

conjc := (conja - conjb) + (conjb - conja)

## END;

## BEGIN

conja := [ ];

conjb := [ ];

read (car);

WHILE car &lt; &gt; '.' DO

## BEGIN

conja := conja + [car];

read (car)

## END;

readln;

read (car);

WHILE car &lt; &gt; '.' DO

## BEGIN

conjb := conjb + [car];

read (car)

## END;

readln;

simétrica (conja, conjb, conjc);

FOR escala := 'A' TO 'Z' DO

IF escala IN conjc

THEN write (escala)

## END.

## 7. TYPE

conjnúm = SET OF 1..128;

FUNCTION miembros [VAR núm : conjnúm] : integer;

## VAR

i, cuenta : integer;

## BEGIN

cuenta := 0;

FOR i := 1 TO 128 DO

IF i IN núm

THEN cuenta := cuenta + 1;

miembros := cuenta

## END

## 8. PROGRAM modif (input, output);

## TYPE

conjdiv 3 SET OF 1..100;

