BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

FACULTAD DE INFORMATICA
U.N.L.P.

Algoritmos, datos y programas

con aplicaciones en Pascal, Delphi y Visual Da Vinci

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

Ing. Armando E. De Giusti
UNIVERSIDAD NACIONAL DE LA PLATA - CONICET

Equipo del LIDI (Laboratorio de Investigación y Desarrollo en Informática) - UNLP

Lic. María C. Madoz
Lic. Rodolfo A. Bertone
Lic. R. Marcelo Naiouf
Lic. Laura C. Lanzarini
C.C. Gladys Gorga
Ms Claudia C. Russo
Lic. Raul Champredonde

Algoritmos, datos y programas

con aplicaciones en Pascal, Delphi y Visual Da Vinci

Prentice Hall

Pearson Educación

Argentina • Bolivia • Brasil • Colombia • Costa Rica • Chile • Ecuador • El Salvador • España • Guatemala • Honduras • México • Nicaragua • Panamá • Paraguay • Perú • Puerto Rico • República Dominicana • Uruguay • Venezuela

Amsterdam • Harlow • Londres • Menlo Park • Munich • Nueva Delhi • Nueva Jersey • Nueva York • Ontario • París • Sidney • Singapur • Tokio • Toronto • Zurich

Datos de catalogación bibliográfica

003.1
DEG

De Giusti, Armando
Algoritmos, datos y programas / Armando de Giusti, María Cristina Madoz y María Cristina Lanzarini. -1ª ed.- Buenos Aires: Pearson Education, 2001.
472 p. ; 25,5 x 19,5 cm.

ISBN 987-9460-64-2

I. Madoz, María Cristina II. Lanzarini, Laura Cristina
1. Programación

COMPRA FACULTAD
$
Fecha 2004
Inv. E. Inv. B. 1546

Editor: Enrique Baumann
Gerente de División: Esteban Lo Presti
Diseño de tapa e interior: Diego Linares
Ilustración de tapa: Claudia Deleau para Línea&Color
Producción: Laura G. Lago - Cristian Rodriguez Tabares

Copyright © 2001 PEARSON EDUCATION S.A.
Av. Regimiento de Patricios 1959 (C1266AAF), Buenos Aires, Rep. Argentina

PRENTICE HALL Y PEARSON EDUCACIÓN son marcas de propiedad de PEARSON EDUCATION S.A.

ISBN: 987-9460-64-2

Primera Edición: Diciembre de 2001
Queda hecho el depósito que dispone la ley 11.723

Este libro no puede ser reproducido total ni parcialmente en ninguna forma, ni por ningún medio o procedimiento, sea reprográfico, fotocopia, microfilmación, mimeográfico o cualquier otro sistema mecánico, fotoquímico, electrónico, informático, magnético, electroóptico, etcétera. Cualquier reproducción sin el permiso previo por escrito de la editorial viola derechos reservados, es ilegal y constituye un delito.

Impreso en Brasil por RR Donnelley, en el mes de diciembre de 2001.
Rua Epiacaba 90 –Vila Arapuá–

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

A todos los alumnos que han pasado por la cátedra Programación de Computadoras desde la creación de la Licenciatura en Informática de la Universidad Nacional de La Plata. Especialmente a aquellos alumnos que han trabajado y trabajan como colaboradores y auxiliares Ad-Honorem en Programación de Computadoras.

A todos los miembros del Laboratorio de Investigación y Desarrollo en Informática (LIDI) que comparten con los autores el trabajo diario de docencia e investigación.

Prólogo

De la experiencia docente al texto universitario

Uno de los desafíos más interesantes de la actividad académica es tratar de escribir un texto universitario destinado a **introducir conceptos básicos para la formación del lector.**

Los autores, con una larga actividad docente y de investigación en Informática, han tratado de superar los límites de la simple **transmisión de información**, buscando iniciar en el lector-alumno un aprendizaje de los fundamentos conceptuales de cada tema.

Este objetivo nos ha obligado a cambiar el estilo habitual de escritura de artículos "científicos" por un planteo más coloquial que conduzca rápidamente a casos concretos que demuestren en forma clara las nociones conceptuales de interés.

A lo largo de más de doce años, los autores han coordinado y dictado cursos básicos de Informática en la Universidad Nacional de La Plata. Precisamente, esta experiencia se traduce en la selección de los temas, la organización y el enfoque de los mismos, la elección de los lenguajes de ejemplificación y las conclusiones de cada capítulo y del texto en general.

La elaboración del libro ha demostrado, una vez más, la riqueza de la cátedra universitaria en la generación de ideas. Los profesores-autores han repasado y discutido interactivamente su enfoque para explicar cada uno de los temas seleccionados ante los alumnos y han tratado de encontrar una síntesis escrita que reflejara del modo más claro posible el **objetivo formativo** buscado.

A lo largo de estos años los autores han realizado numerosas publicaciones relacionadas con la temática del texto y han elaborado herramientas especialmente orientadas a un primer nivel formativo en Programación. Tal vez la más destacada que se incorpora en la ejemplificación de este libro es el ambiente y lenguaje de programación Visual Da Vinci, actualmente utilizada en diferentes Universidades del país y del exterior.

Creemos que la elaboración del texto en sí ha actuado como un estímulo para formalizar y sintetizar los conceptos esenciales de un curso inicial de Algorítmica y Estructuras de Datos, objetivo que supera el de su utilización en la cátedra de Programación de Computadoras de la Facultad de Informática de la UNLP.

La temática y su inclusión en las carreras de Informática

La introducción de los temas de especificación y expresión de algoritmos, combinados con el tratamiento de las estructuras de datos básicas de los lenguajes de programación, constituye el núcleo de la mayoría de los textos iniciales de Informática. En general, el paradigma de programación es procedural, con énfasis en la modulación y la programación estructurada. En ese aspecto, este libro sigue la línea de Wirth, buscando potenciar el tratamiento de los algoritmos con las nociones de eficiencia-corrección y, al mismo tiempo, extender los conceptos de tipos de datos a tipos definidos por el usuario y tipos abstractos de datos.

Entre los aportes nuevos se cuenta con un Anexo dedicado al lenguaje y ambiente Visual Da Vinci (el cual puede obtenerse libremente en Internet) y dos capítulos asociados con la programación orientada a eventos y al lenguaje Delphi, que también se pueden completar con un curso específico ejemplificado que está disponible en Internet.

Objetivos, alcance y enfoque de este texto

Este libro tiene tres ejes fundamentales: la especificación y expresión de algoritmos, analizando su corrección y eficiencia; el tratamiento de estructuras de datos simples y compuestas hasta llegar al concepto de tipo abstracto de datos, y la contextualización de la expresión de algoritmos en el ámbito más amplio del desarrollo de los sistemas de software, es decir, de la Ingeniería de Software.

En el enfoque se trata de combinar el tratamiento algorítmico con la elección de las estructuras de datos, sin dejar de lado aspectos requeridos por la Ingeniería de Software, como lo son la documentación y la reusabilidad. Para esto se ha elegido un lenguaje básico de ejemplificación como Pascal, combinándolo (según el tema en tratamiento) con el lenguaje elemental

de un robot móvil Visual Da Vinci y en los capítulos finales con Delphi (utilizado para la introducción a la programación orientada a eventos). Cuando ha sido necesario también se ha recurrido a ADA, para tratar de dar consistencia a los conceptos con la herramienta de implementación.

El objetivo formativo, basado en un enfoque concreto de casos que muestran los conceptos en la programación de algoritmos, conduce a que en el texto se intercalen los temas puramente algorítmicos (estructuras de control, recursividad, eficiencia, verificación) con el tratamiento de diferentes estructuras de datos (estáticas, dinámicas, recursivas) tratando de resolver distintas clases de problemas en base a diferentes enfoques algorítmicos, con diferentes estructuras de datos: se trata que el lector-alumno asimile el concepto esencial que un especialista en Informática debe ser un Analista del mundo real, capacitado para elegir alternativas de solución utilizando computadoras en función del problema a resolver, del contexto de dicho problema en el ambiente real, de los recursos disponibles y de las restricciones concretas de implementación de la solución.

La expectativa de los autores

Los autores consideran que este texto (que es una evolución del material de cátedra que han publicado desde 1996, incluyendo una primera versión del libro editada por la Fundación Ciencias Exactas) es un aporte que naturalmente se enriquecerá con las correcciones, modificaciones, sugerencias y extensiones que propongan docentes y alumnos que lo utilicen.

De ningún modo vemos el texto como algo definitivo o consolidado, sino como una expresión ordenada de la experiencia docente de los autores en un tema de formación básica en Informática.

Como todas las Ciencias, la Informática tal como la hemos conocido y tal como la tratamos hoy, está cambiando: la transformación se refleja en las metodologías, herramientas, tecnologías e, incluso, en la formación de recursos humanos en la disciplina.

Por todo esto creemos que este texto está destinado a cambiar, a transformarse y actualizarse, acompañando el cambio en la Ciencia Informática.

Ing. Armando E. De Giusti

Lic. María C. Madoz - Lic. Rodolfo A. Bertone
Lic. R. Marcelo Naiouf - Lic. Laura C. Lanzarini
C.C. Gladys Gorga - Ms Claudia C. Russo
Lic. Raul Champredonde

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

Tabla de contenidos

Capítulo 1

Conceptos básicos 1

1.1 Definiciones 4
1.2 Modelización de problemas del mundo real 8
1.2.1 Otros modelos conocidos del mundo real 11
1.2.2 La especificación de los problemas del mundo real 13
1.2.3 La expresión de soluciones ejecutables en una computadora 14
1.3 Del problema real a su solución por computadora 15
1.3.1 Importancia del contexto 16
1.3.2 Descomposición de problemas. Concepto de modularización 16
1.3.3 Concepto de algoritmo 16
1.3.4 Las partes de un programa 17
1.4 Software 18
1.4.1 Etapas en la resolución de problemas con computadora 19
Conclusiones 20

Capítulo 2

Algoritmos 21

2.1 Estructuras de control 24
2.1.1 Estructuras de control básicas 24
2.1.2 Estructura esquemática de un programa 36
2.2 Importancia de la documentación de un algoritmo 38
2.3 Correción de algoritmos. Importancia de la verificación 38
2.4 Eficiencia de un algoritmo 40
2.5 Descomposición de problemas 41
2.5.1 Modularización 41
2.5.2 Alcance de los datos 48
Conclusiones 60

Capítulo 3

Tipos de datos simples 61

3.1 Tipos de datos 64
3.1.1 Tipo de dato numérico 64

3.1.2 Tipo de dato lógico 68
3.1.3 Tipo de dato carácter 69
3.2 Constantes y variables 71
3.2.1 Declaraciones 71
3.2.2 Asignaciones 73
3.3 Funciones predefinidas 74
3.3.1 Funciones sobre los tipos de datos numéricos 74
3.3.2 Funciones sobre el tipo de datos carácter 76
3.4 Tipos ordinales 77
3.4.1 Funciones sobre tipos de datos ordinales 78
3.5 Tipos de datos 79
3.5.1 Declaración de tipos 80
3.5.2 Tipos, asignaciones y subprogramas 82
3.6 Tipos de datos definidos por el usuario 83
3.6.1 Tipo de dato enumerativo 83
3.6.2 Tipo de dato subrango 86
3.6.3 Tipo de dato conjunto 87
3.6.4 Tipo de dato string 91
Conclusiones 97
Ejercicios 97

Capítulo 4

Procedimientos y funciones. Parámetros 99
4.1 Subprogramas o módulos 102
4.1.1 Alcance de un módulo 103
4.2 Procedimientos 105
4.3 Funciones 106
4.4 Parámetros 107
4.4.1 Correspondencia entre parámetros actuales y formales 107
4.4.2 Métodos para el pasaje de parámetros 108
4.4.3 Implementación de pasaje de parámetros de Pascal 109
4.4.4 Implementación de pasaje de parámetros de Visual Da Vinci 113
4.5 Variables locales y variables globales 113
4.6 Procedimientos y Funciones con parámetros 114
Conclusiones 117

Capítulo 5

Estructuras de datos compuestas 119
5.1 Introducción 122
5.2 Registros 123
5.2.1 Declaración de registros 124

BIBLIOTECA
FAC. DE INFORMATICA
U.N.L.P.

5.2.2 Acceso a los campos de un registro 126
5.2.3 Anidamiento de registros 126
5.2.4 Operaciones sobre registros 127
5.2.5 La sentencia with 129
5.3 Pilas 132
5.3.1 Declaración de pilas 133
5.3.2 Operaciones sobre pilas 133
5.4 Colas 140
5.4.1 Declaración de colas 141
5.4.2 Operaciones sobre colas 142
5.5 Concepto de tipo definido por el usuario. Extensiones a pilas y colas 149
Conclusiones 150
Ejercicios 150

Capítulo 6

Datos compuestos indexados: arreglos 153
6.1 Clasificación de las estructuras de datos 156
6.2 Arreglos 157
6.2.1 Vectores 157
6.2.2 Matrices 167
6.2.3 Arreglos n-dimensionales 174
6.2.4 Arreglos como parámetros 176
6.3 Comparación de estructuras de datos arreglo con pilas y colas 177
6.3.1 Resultado de la ejecución del algoritmo anterior 179
Conclusiones 180
Ejercicios 181

Capítulo 7

Recursividad 183
7.1 Recursividad 186
7.1.1 Aspectos a tener en cuenta en una solución recursiva 186
7.1.2 Casos de estudio 188
7.2 Ejecución de un programa y la pila de activación 192
Conclusiones 195
Ejercicios 195
Referencias bibliográficas 196

Capítulo 8

Análisis de algoritmos: concepto de eficiencia 197
8.1 El concepto de eficiencia 200
8.2 Análisis de eficiencia de un algoritmo 200

8.3 Análisis de algoritmos según su tiempo de ejecución 201
8.3.1 ¿Qué medir? 201
8.3.2 Programas eficientes 203
8.3.3 Tiempo de ejecución de un algoritmo 204
8.4 Análisis de algoritmos según su aprovechamiento de memoria 208
8.5 Eficiencia en algoritmos recursivos 208
8.6 Estimando el tiempo de ejecución de un algoritmo recursivo 209
8.7 Algoritmo de búsqueda 210
8.7.1 Introducción 210
8.7.2 Búsqueda Lineal 212
8.7.3 Búsqueda binaria 214
8.7.4 Análisis de los algoritmos de búsqueda 216
8.8 Algoritmos de ordenación 220
8.8.1 Consideraciones generales 221
8.8.2 Método de Selección 221
8.8.3 Método de intercambio o Burbujeo 223
8.8.4 Método de inserción 225
8.8.5 Análisis de los métodos de ordenación elementales 226
8.8.6 Análisis del peor caso de los algoritmos de ordenación 228
8.9 Ordenación por índices 230
8.10 Métodos de ordenación eficientes 236
8.10.1 Sorting by Merging 236
8.11 Recursividad y eficiencia 245
Conclusiones 246
Ejercicios 246

Capítulo 9

Datos compuestos enlazados: listas, árboles y grafos 247
9.1 Listas como estructura de datos 250
9.1.1 Punteros 251
9.1.2 Operaciones con listas 254
9.1.3 Conclusiones sobre listas 258
9.2 Árboles 259
9.2.1 Representación de árbol binario 261
9.2.2 Conclusiones 262
9.3 Grafos 262
9.3.1 Representación de Grafos 266
9.3.2 Buscando un camino entre dos vértices del grafo 268
Conclusiones 271
Ejercicios 271

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

Capítulo 10

Introducción a tipos abstractos de datos 273
10.1 Abstracciones de datos 276
10.2 Conceptos sobre tipos de datos 276
10.2.1 Sistema de Tipos 277
10.3 Módulos, interfaz e implementación 278
10.4 Encapsulamiento de datos 279
10.5 Diferencia entre tipo de dato y tipo abstracto de dato 281
10.5.1 Ventajas del uso de TAD respecto de la programación convencional 286
10.5.2 Formas de abstracción: tipos de datos abstractos y tipos de datos lógicos 286
10.6 Requerimientos y diseño de un TAD 287
Conclusiones 288
Ejercicios 288

Capítulo 11

Análisis de algoritmos 289
11.1 Repaso a conceptos básicos del análisis de algoritmos 292
11.1.1 Clasificación de algoritmos 292
11.1.2 Análisis de caso promedio y caso peor 293
11.2 Relaciones básicas de Recurrencia 294
11.2.1 Caso 1: Ley CN = CN-1 + N 294
11.2.2 Caso 2: Ley CN = CN/2 + 1 295
11.2.3 Caso 3: Ley CN = CN/2 + N 295
11.2.4 Caso 4: Ley CN = 2CN/2 + N 295
11.2.5 Caso 5: Ley CN = 2CN/2 + 1 296
11.3 Soluciones recursivas y no recursivas 296
11.3.1 Caso 1: La serie de Fibonacci 297
11.3.1 Caso 2: La construcción de una regla graduada 298
11.3.3 Caso 3: Construcción de un fractal 300
11.3.4 Caso 4: Soluciones recursivas y no-recursivas en el ejemplo de árboles ... 302
11.3.5 Transformación de soluciones recursivas en no recursivas 310
11.4 Soluciones de algoritmos con estructuras de datos estáticas o dinámicas 310
11.4.1 Colas 311
11.5 Análisis de algoritmos no numéricos: tratamiento de strings 314
11.5.1 Algoritmo trivial (fuerza bruta, "*Brute Force*") 315
11.5.2 Algoritmo de Knuth-Morris-Pratt 317
11.5.3 Algoritmo de Boyer-Moore 321
Conclusiones 323
Referencias bibliográficas 323

Capítulo 12

Introducción al concepto de archivos 325
12.1 Conceptos Generales 328
12.1.2 Tiempos de acceso 328
12.1.3 Archivos 329
12.1.4 Administración de archivos 329
12.1.5 Operaciones básicas sobre archivos 330
12.1.6 Técnicas de organización y acceso a un archivo 331
12.1.7. Manejo de buffers 332
12.2 Operaciones básicas sobre archivos en Pascal 333
12.2.1 Declaración de archivos 333
12.2.2 Relación con el sistema operativo 334
12.2.3 Apertura y creación de archivos 335
12.2.4 Cierre de archivos 335
12.2.5 Lectura y escritura en archivos 335
12.2.6 Operaciones adicionales 336
12.3 Algoritmos clásicos sobre archivos 338
12.3.1 Caso 1: Modificación de datos de un archivo 338
12.3.2 Caso 2: Agregado de datos a un archivo 340
12.3.3 Caso 3: Actualización de un archivo maestro con uno o varios archivo detalle 341
12.3.4 Caso 4: El problema del "corte de control" 346
12.3.5 Caso 5: El problema de la unión de archivos (merge) 349
12.4 Eliminar elementos de un archivo 354
12.4.1 Borrado físico 354
12.4.2 Borrado lógico 355
12.4.3 Estrategia de trabajo 355
12.5 Nociones generales sobre bases de datos 355
12.5.1 Archivos secuenciales indizados 355
12.5.2 Bases de datos 358
Conclusiones 359
Ejercicios 359

Capítulo 13

Conceptos de metodologías en el desarollo de sistemas de software 361
13.1 Ingeniería de Software 364
13.2 El ciclo de vida clásico de un sistema de software 364
13.2.1 Análisis y Especificación de Requerimientos 365
13.2.2 Diseño y especificación del sistema 365
13.2.3 Codificación y verificación de los módulos 365

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

13.2.4 Integración y prueba global del Sistema 366
13.2.5 Mantenimiento del Sistema 366
13.2.6 Tiempos y costos requeridos por las etapas del ciclo de vida 367
13.3 Especificación, codificación y prueba de algoritmos 368
13.4 El método de refinamientos sucesivos 369
13.5 La noción de paradigma de programación 369
13.6 El rol del especialista en software 370
13.7 Aspectos importantes de los sistemas de software 370
13.7.1 Corrección 371
13.7.2 Verificabilidad 371
13.7.3 Confiabilidad 371
13.7.4 Eficiencia 371
13.7.5 Interfaz amigable 372
13.7.6 Facilidad de mantenimiento 372
13.7.7 Reusabilidad 372
13.7.8 Portabilidad 372
13.8 Principios generales de la Ingeniería de Software 373
Conclusiones 373
Referencias bibliográficas 374

Capítulo 14

Programación orientada a eventos 375
14.1 Programación orientada a eventos 378
Conclusiones 383

Capítulo 15

Introducción a la programación en Delphi 385
15.1 Introducción 388
15.2 Nociones básicas 388
15.3 Propiedades y Eventos 389
15.4 Eventos y métodos 390
15.5. Esquema de un programa Delphi 391
15.6 El primer ejemplo 392
15.6.1 Creando un nuevo proyecto 392
15.6.2 Ejecución del programa 393
15.6.3 Incorporación de componentes 394
15.6.4 Modificación de propiedades 394
15.6.5 Agregando reacciones a eventos 395
15.7 Aspectos generales de las componentes de Delphi 398

15.7.1 Propiedades más comunes 398
15.7.2. Eventos más comunes 399
15.8 Entrada/salida en Delphi 399
15.8.1 Componentes básicos 400
15.8.2 Procesos predefinidos que permiten visualizar un string: 400
15.9 Primer ejemplo utilizando entrada/salida 402
15.10 Manejo de excepciones 404
15.11 Analizando la unidad que maneja el formulario 408
15.12 Compartiendo eventos. El uso del parámetro Sender 411
Conclusiones 414

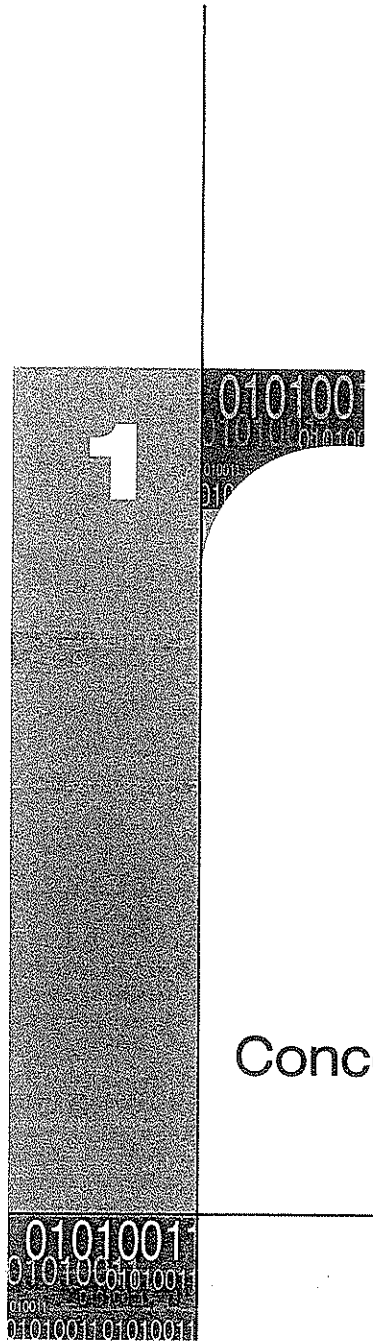
Conclusiones 415

Apéndice A 417

Apéndice B 437

Apéndice C 441

Índice analítico 447



Capítulo 1

Conceptos básicos

1

Conceptos básicos

Objetivo

En este capítulo se presentan algunas definiciones básicas que servirán de conceptos introductorios. Muchas de ellas son conocidas total o parcialmente por el lector y se discutirán en el contexto más amplio de software y sistemas de software aplicados a la resolución de problemas en el mundo real.

El eje temático gira alrededor del concepto de modelo como representación del ambiente, los datos y los procesos en el mundo real y de la noción de programa como expresión ordenada, completa y correcta de la especificación de una solución (algoritmo) computable de dicho problema.

Simultáneamente con las definiciones rigurosas, se presentan algunos ejemplos sencillos a fin de que el lector reflexione sobre el modo de abstraer un problema del mundo real, convertirlo en un modelo computable y escribir una solución simbólica del mismo. En particular, se trabaja con una máquina abstracta simple (robot móvil controlado por un conjunto elemental de primitivas de control) que permite modelizar recorridos y tareas.

1.1 Definiciones

Definición

La informática es la ciencia que estudia el análisis y resolución de problemas utilizando computadoras.

La palabra ciencia se relaciona con una metodología fundamentada y racional para el estudio y resolución de los problemas. En este sentido la Informática se vincula especialmente con la Matemática y la Ingeniería.

La resolución de problemas utilizando las herramientas informáticas puede tener aplicaciones en áreas muy diferentes tales como biología, comercio, control industrial, administración, robótica, educación, arquitectura, diseño, etc.

Los temas propios de la ciencia Informática abarcan aspectos tales como la arquitectura física y lógica de las computadoras, las metodologías de análisis y diseño de sistemas de software, los lenguajes de programación, los sistemas operativos, la inteligencia artificial, los sistemas de tiempo real, el diseño y aplicación de bases de datos, etc.

Definición

Una computadora es una máquina digital y sincrónica, con cierta capacidad de cálculo numérico y lógico, controlada por un programa almacenado y con posibilidad de comunicación con el mundo exterior.

En esta definición (que es incompleta y perfectible, pero sirve a los fines introductorios de este capítulo) se establece que una computadora moderna es esencialmente digital; esto significa que las señales eléctricas que maneja, y la información que procesa se representa en forma discreta, normalmente con valores binarios (0 o 1).

Además, se afirma que es sincrónica, es decir, que realiza las operaciones coordinadas por un reloj central que envía pulsos de sincronismo a todos los elementos que componen la computadora. Esto significa que todas las operaciones internas se realizan en instantes de tiempo predefinidos y coordinados con el reloj.

Internamente las computadoras tienen cierta capacidad de cálculo numérico y lógico, en un subsistema conocido como Unidad Aritmético-Lógica (UAL). Normalmente, las operaciones que pueden realizarse en ella son muy simples (suma, or, and, comparaciones).

La frase "controlada por un programa" es quizás la más importante de esta definición. Significa que internamente se tienen órdenes o instrucciones almacenadas, que la computadora podrá leer, interpretar y ejecutar ordenadamente.

Por último, se requiere que la computadora pueda vincularse con el mundo real (por ejemplo, a través de un teclado como entrada de información y una pantalla como salida de la misma).

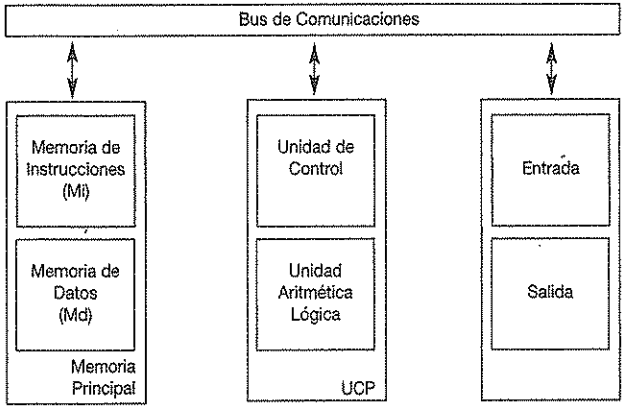


Figura 1.1

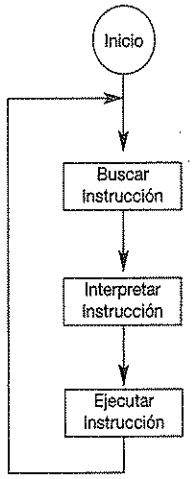


Figura 1.2

El funcionamiento de una computadora. representada esquemáticamente por el modelo de la Figura 1.1, se puede sintetizar en el esquema de la Figura 1.2 que representa una secuencia infinita de pasos:

- Buscar la próxima instrucción a ejecutar Ii de la memoria de instrucciones Mi.
- Interpretar qué hacer con Ii en la Unidad de Control (UC).
- Ejecutar las operaciones interpretadas por UC, utilizando la UAL de ser necesario. Estas operaciones pueden comprender lectura/escritura de la memoria de datos Md o entrada/salida por los periféricos Pe o Ps.

En la Figura 1.1 se ha dividido conceptualmente la memoria central M en memoria de instrucciones (Mi) donde residen las órdenes que la computadora debe interpretar y ejecutar y memoria de datos (Md) donde se almacena la información con la cual la computadora realizará los procesos (cálculos, decisiones, actualizaciones) que sean necesarios para la resolución del problema.

El bloque marcado como entrada/salida (e-s) en la Figura 1.1 representa los dispositivos que permiten la comunicación con el mundo real. Por ejemplo, el controlador de video que vincula el procesador central de la computadora con la pantalla o el circuito controlador de multimedia que permite tener salida por un parlante o entrada por un micrófono.

Las líneas de comunicación indicadas como bus de comunicación (BC) normalmente permiten el paso de tres grandes categorías de información: direcciones, datos y control. El lector puede consultar algunos textos relacionados con la arquitectura física de las computadoras tales como (Stallings, 1993) y (Tanenbaum, 1988) para interiorizarse de los diferentes tipos de buses, sus características físicas y su importancia en el funcionamiento de una computadora. En el esquema simplificado se acepta que estas líneas permiten la comunicación interna y externa de datos, direcciones y señales de control.

Por último, tradicionalmente la combinación de la UC y la unidad de cálculo, UAL, se llama unidad central de procesamiento (UCP), que en las computadoras personales está representada por el microprocesador (por ejemplo, 486, 586, Pentium, etc.).

Definición

Un programa es un conjunto de instrucciones, ejecutables sobre una computadora, que permite cumplir una función específica.

En esta definición se asocia al programa con una función determinada o requerimiento a satisfacer por la ejecución del conjunto de instrucciones que lo forman.

Ejemplos de programa son aquellos que permiten emitir los cheques de pago de sueldos de una empresa o aquel que lleva a que un robot recorra las calles de una ciudad recogiendo al

menos una flor de cada esquina donde hubiera (Ver Apéndice A). En el Ejemplo 1.1 se muestra el programa que resuelve este problema.

Normalmente, los programas alcanzan su función objetivo en un tiempo finito. Sin embargo, se pueden tener aplicaciones (por ejemplo, el programa de control de un sistema de alarmas) que se ejecutan indefinidamente, porque poseen un requerimiento de tiempo infinito.

Un programa sin errores que se ejecuta sobre una computadora, que comienza y termina, puede ser incorrecto si no cumple con lo solicitado en los requerimientos. Por ejemplo, si el robot debe recorrer todas las esquinas de la ciudad recogiendo todas las flores que encuentre, el ejemplo 1.1 es una solución incorrecta pues la instrucción `tomarFlor` se ejecuta a lo sumo una vez para cada esquina. En este caso deben reemplazarse las instrucciones selectivas por instrucciones iterativas condicionales (`si por mientras`).

Ejemplo 1.1

```
programa recorrido
comenzar
   iniciar
   derecha
   repetir 100
      repetir 99
         si HayFlorEnLaEsquina
            tomarFlor
         mover
      si HayFlorEnLaEsquina
         tomarFlor
      si PosCa < 100
         Pos(1, PosCa+1)
fin
```

(este ejemplo es sintácticamente correcto, pero no cumple con lo pedido en el enunciado)

Definición

Un dato es una representación de un objeto del mundo real mediante la cual se pueden modelizar aspectos de un problema que se desea resolver con un programa sobre una computadora.

Es muy difícil representar datos reales en una computadora (que hemos definido como una máquina digital binaria, es decir, solo con capacidad de almacenar ceros y unos). Imaginemos un objeto como las flores de la ciudad, que el robot recoge. ¿Qué son dentro de la computadora? Seguramente algo menos real que una flor verdadera.

Representar datos, aun los más simples como los dígitos decimales, una letra, un nombre o un color requiere una transformación desde el mundo real a alguna forma de representación binaria que pueda ser interpretada por la computadora.

Datos más complejos como una imagen, una canción o la trayectoria de un avión también son representados en forma binaria. Sin embargo, para establecer la forma de modelizarlos e interpretarlos será necesario un análisis cuidadoso por parte de quien escriba un programa que los utilice.

A priori las dos tareas más importantes con las que se enfrenta quien debe escribir un programa para resolver un problema sobre una computadora son:

a) definir el conjunto de instrucciones cuya ejecución ordenada conduce a la solución

b) elegir la representación adecuada de los datos del problema.

En síntesis, las computadoras son una poderosa herramienta para la resolución de problemas, cuya potencial utilidad depende de la capacidad de programación de soluciones adecuadas a cada problema.

La función esencial del especialista informático es explotar la potencialidad de las computadoras (velocidad, exactitud, confiabilidad) para resolver situaciones del mundo real.

Para esto debe analizar los problemas del mundo real, ser capaz de sintetizar sus aspectos principales y poder especificar la función objetivo que se desea. Posteriormente, debe expresar la solución en forma de programa, manejando los datos del mundo real mediante una representación válida para una computadora.

1.2 Modelización de problemas del mundo real

El mundo real es naturalmente complejo y en muchas ocasiones los problemas a resolver resultan difíciles de sintetizar. Imaginemos, por ejemplo, el caso de incorporar una computadora en un comercio cualquiera. Posiblemente el dueño entienda que su objetivo es aumentar las ganancias del negocio a través de la inversión que significa incorporar una máquina y un sistema de software.

Sin embargo, no se puede asegurar que esta inversión se traduzca directamente en mayor rentabilidad. Tendremos que analizar cuáles aspectos de la operatoria del negocio (emisión de remitos y facturas, control de cuentas corrientes, seguimiento de cobros a clientes, liquidación de impuestos, control de bancos, etc.) pueden requerir una automatización y en qué casos y de qué forma esta automatización puede traducirse en un beneficio (en tiempo, en empleados, en calidad del servicio al cliente, en confiabilidad, etc.).

Definición

El proceso de análisis del mundo real para interpretar los aspectos esenciales de un problema y expresarlo en términos precisos se denomina abstracción.

Definición

Abstraer un problema del mundo real y simplificar su expresión, tratando de encontrar los aspectos principales que se pueden resolver (requerimientos) los datos que se han de procesar y el contexto del problema se denomina modelización.

Un modelo simple que se discute en el Apéndice A es el del robot y la ciudad en la que se mueve. El problema del mundo real (recorrer una ciudad real con un robot que realiza determinadas tareas controlado por un programa) es muy complejo (imagine el lector que solo el subproblema de tener un sistema de visión por computadora en un robot, que le permita reconocer un objeto determinado, lleva años de estudio y miles de líneas de instrucciones para ejecutarse).

En el Apéndice A se busca modelizar el aspecto esencial de interés: escribir programas que permitan al robot realizar recorridos con determinados objetivos (limpiar la ciudad de papeles, recoger o depositar flores, sortear obstáculos, etc.).

En la búsqueda de reducir el problema del mundo real a los aspectos básicos que debe cubrir el robot en el ambiente Visual Da Vinci se han realizado las siguientes abstracciones:

- La ciudad queda reducida a un ámbito rectangular de calles y avenidas.
- El andar del robot queda asociado con 1 paso = 1 cuadra de recorrido.
- Se reducen los datos en el modelo para tratar solo con flores, papeles y obstáculos.
- Se aceptan convenciones (el robot solo inicia sus recorridos en la posición (1,1) de la ciudad)
- Se supone que el robot ve y reconoce los objetos de interés, sin discutir cómo lo puede hacer.

Es interesante discutir el grado de exactitud del modelo y su relación con los objetivos a cumplir. Está claro que en este ejemplo no se modelizan exactamente la ciudad ni los objetos que están en ella. Tampoco se representa adecuadamente el movimiento de un robot real, que posiblemente tenga que dar un número grande de pasos para recorrer una cuadra. Se ignoran los detalles del proceso de reconocimiento de los objetos e, incluso, no se considera la posibilidad de que el robot confunda objetos.

Sin embargo, dado que el objetivo planteado es escribir programas que permitan representar recorridos con funciones simples (contar, limpiar, depositar) el modelo esencial es suficiente y funciona correctamente.

De hecho, se puede refinar y ajustar fácilmente el modelo, corrigiendo, por ejemplo, el proceso de avanzar una cuadra, reemplazándolo por otro en el que el robot da los *n* pasos necesarios para hacer esa cuadra.

A partir de un modelo esencial del problema a resolver podemos analizar sus diferencias con respecto a los requerimientos definidos, ajustando el modelo, si fuera necesario.

Siguiendo con la presentación del robot, se ha visto el contexto (la ciudad) en el cual se desplaza y alguno de los datos a tratar (por ejemplo, flores y papeles).

Es interesante ahora discutir las órdenes que puede ejecutar.

Cada una de las acciones corresponde a una instrucción, entendible por el robot y que debe tener un modo unívoco de expresión, para que la máquina la interprete correctamente, y un significado único, a fin de poder verificar que el resultado final de la tarea se corresponde con lo requerido.

De esta manera, camina, recoge, cuenta, deposita y puede contestar si su bolsa está vacía. Este conjunto de instrucciones elementales que se detallan en el Apéndice A permiten escribir programas con un objetivo bien definido, que tendrán una interpretación y una ejecución única por el robot.

Ejemplo 1.2

El robot está ubicado inicialmente en la esquina correspondiente a la calle 4 y la avenida 4 de la ciudad. En su bolsa no tiene flores, pero sí 10 papeles. Se quiere desarrollar un programa que le permita recorrer la calle 4 hasta la avenida 50, depositando un papel por esquina mientras tenga papeles en la bolsa. En caso que en las esquinas hubiera flores, debe recogerlas y ponerlas en su bolsa. El ejemplo 1.2 muestra la resolución de este problema.

```
programa calle4
comenzar
  iniciar
  Pos(4,4)
  {El robot está ubicado en (4,4), no tiene flores en la bolsa y
   tiene 10 papeles}
  derecha
  mientras ( PosAv < 50 )
    si HayPapelEnLaBolsa
      depositarPapel
    mientras HayFlorEnLaEsquina
      tomarFlor
    mover

  {analizar la esquina (4,50)}
  si HayPapelEnLaBolsa
    depositarPapel
  mientras HayFlorEnLaEsquina
    tomarFlor
  {El Robot queda ubicado en (4,50), sin papeles en la bolsa}
fin
```


Con esta solución, y de acuerdo a lo especificado en el enunciado, ¿cómo se distribuyen los papeles en el recorrido?

Hasta ahora se han definido instrucción, programa y dato. Es interesante discutir dos términos que aparecerán habitualmente en la escritura de programas: precondición y poscondición.

Definición

Una precondición es una información que se conoce como verdadera antes de iniciar el programa.

Por ejemplo, se ha supuesto que al iniciar el recorrido en el Ejemplo 1.2, el robot estaba ubicado en la posición (4,4) de la ciudad con la bolsa conteniendo 0 flores y 10 papeles.

Definición

Una poscondición es una información que debiera ser verdadera al concluir un programa, si se cumple adecuadamente el requerimiento pedido.

En el ejemplo anterior, debería terminar el recorrido con el robot en la posición (4,50) de la ciudad con la bolsa conteniendo 0 papeles. Con respecto a las flores, no puede realizarse ninguna afirmación. Queda para el lector este análisis.

1.2.1 Otros modelos conocidos del mundo real

El lector debe conocer numerosas modelizaciones del mundo real que ha estudiado en la escuela primaria o secundaria. Se mencionan brevemente tres de ellas, recordando las inexactitudes que se aceptan del modelo correspondiente:

- En el Tiro Oblicuo (Figura 1.3) se supone que el objeto es lanzado al vacío (es decir, sin tener en cuenta la resistencia del aire); no se considera la posibilidad de que haya viento en cualquier dirección; se considera la fuerza de la gravedad constante y no se discuten posibles cambios de altitud en el terreno. Las fórmulas que caracterizan el alcance y la altura máxima son esencialmente correctas, aunque si se realizan mediciones reales utilizando, por ejemplo, un cañón y proyectiles, posiblemente haya errores menores en el alcance de los mismos y en la altura máxima alcanzada.

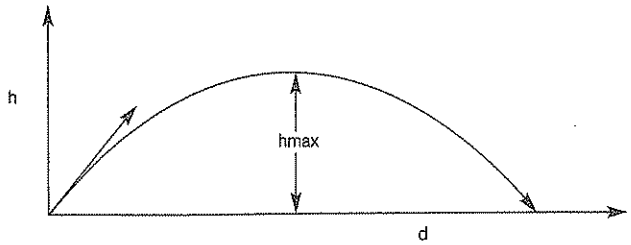


Figura 1.3

- En el movimiento rectilíneo uniformemente acelerado, Figura 1.4, se supone que el móvil no sufre la resistencia del aire; que el terreno es perfectamente horizontal; que no hay una parte de la fuerza que se "consume" en vencer el rozamiento del piso y que la forma del móvil no tiene importancia. Las curvas de recorrido y velocidad que se presentan en las Figuras 1.5 y 1.6 son conceptualmente correctas, pero deberían corregirse si, por ejemplo, nuestro móvil se desliza sobre arena o en pendiente.

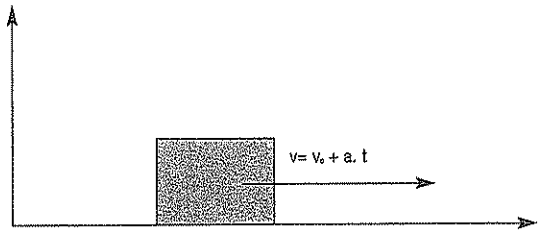


Figura 1.4

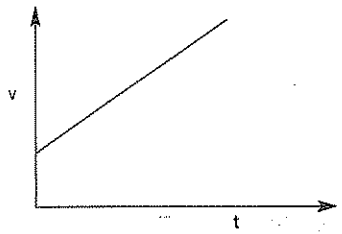


Figura 1.5

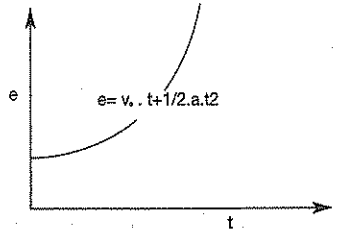


Figura 1.6

Por último, cuando se estudian problemas de choque de objetos, Figura 1.7, generalmente se ignora la deformación que provoca el choque en esos objetos y el gasto de energía que ello significa. En el modelo ideal estudiado en Física, las bolas de billar no se detendrían nunca; sin embargo, sabemos que sí lo hacen debido al consumo de energía de los choques y al rozamiento sobre el paño de la mesa de billar. Nuevamente, el modelo esencial sirve para comprender lo básico del problema y si se pretende una mayor exactitud se debe refinar y ajustar, acercándolo al mundo real, a costo de incrementar la complejidad.

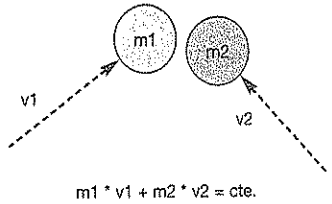


Figura 1.7

1.2.2 La especificación de los problemas del mundo real

Hasta aquí se ha expresado que los problemas del mundo real son complejos y que, por ese motivo, es necesario buscar sus aspectos esenciales para representarlos mediante un modelo. Además, los objetivos buscados al enfrentar el problema deben ser expresados rigurosamente.

Definición

El proceso de analizar los problemas del mundo real y determinar en forma clara y concreta el objetivo que se desea se denomina especificación.

Especificar un problema significa establecer en forma unívoca el contexto, las precondiciones y el resultado esperado, del cual se derivan las poscondiciones.

Naturalmente, para tener especificaciones unívocas es deseable poder expresar los problemas en un lenguaje correcto, formalmente riguroso y que no permita múltiples interpretaciones. Siempre se debe pensar que el modelo debe ser interpretado sin errores por un autómata, como una computadora. Lo ideal sería poder reducir los problemas del mundo real a fórmulas matemáticas, como en alguno de los modelos de la Física que hemos mencionado. Sin embargo las áreas de aplicación de la informática no siempre lo permiten. Las aproximaciones serán notaciones variadas (gráficas, algebraicas, funcionales, textuales) que tratarán de representar el análisis del problema en forma unívoca.

La calidad de la especificación condiciona todo el proceso de desarrollo de soluciones por computadora. Normalmente, el mundo y el lenguaje del usuario, quien utilizará el programa desarrollado, son muy diferentes del mundo del especialista en informática que desarrolla el programa. El punto de contacto básico es el análisis del problema y su traducción desde el mundo real (usuario) al modelo abstracto que podrá resolverse sobre una computadora. La exactitud con la cual esta traducción aproxima el modelo a los objetivos del usuario condiciona fuertemente el resultado final del sistema.

1.2.3 La expresión de soluciones ejecutables en una computadora

Así como la especificación es fundamental para convertir el problema real en un modelo que permita la utilización de computadoras en su solución, la escritura de un programa que represente una solución ejecutable constituye la piedra basal de la Ciencia de la Computación.

Este texto estará dedicado primordialmente a la expresión de soluciones como programas. Se presentó anteriormente que cada acción primitiva a ejecutar en una computadora corresponde a una instrucción, y que cada instrucción debe tener un modo único de escribirse (sintaxis) y un modo único de interpretarse y ejecutarse (semántica) para poder ser utilizadas por una computadora.

Definición

El conjunto de instrucciones permitidas y definidas por sus reglas sintácticas y su valor semántico, para la expresión de soluciones a problemas, constituye un lenguaje de programación.

El ejemplo primario de lenguaje de programación es el pequeño conjunto de instrucciones que permite al robot realizar recorridos por la ciudad. Este lenguaje se denomina Visual Da Vinci y se halla descrito en el Apéndice A.

En adelante, los ejemplos de este texto serán resueltos utilizando diferentes lenguajes. La riqueza expresiva y la legibilidad de las primitivas de un lenguaje de programación serán muy importantes para escribir programas que respondan a lo especificado.

1.3 Del problema real a su solución por computadora

La Figura 1.7 muestra los procesos analizados conceptualmente en los párrafos anteriores:

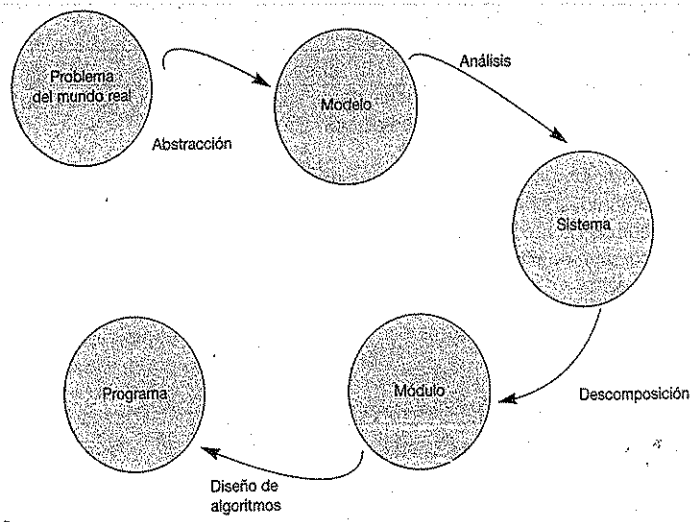


Figura 1.8

- Para llegar al modelo, a partir del problema se debe realizar un proceso de abstracción.
- A partir del modelo se debe elaborar el análisis de la solución como sistema. Esto significa una descomposición en módulos y un estudio de los datos del problema.
- Cada uno de los módulos debe tener un proceso de refinamiento para expresar su resolución en forma ordenada, lo que llevará a construir los algoritmos correspondientes. Es importante destacar que no siempre un problema puede tener una solución algorítmica.
- A partir de los algoritmos se pueden escribir y probar programas en un lenguaje determinado y con datos representativos del problema.
- Es importante comprender que llegar a un modelo no significa que el problema sea computable, es decir, resoluble por computadora.
- Por ejemplo, se puede tener un algoritmo perfecto para calcular números primos. De hecho el lector podría seguramente escribir un programa para computar un número finito de números primos en un rango determinado. Sin embargo, si se quisieran calcular todos los números primos, no habría una solución computable en un tiempo finito.

1.3.1 Importancia del contexto

El contexto del problema real, (por ejemplo, la ciudad en la que se mueve el robot o el modo de operación en la atención de clientes de un comercio) es muy importante para analizar y diseñar la solución utilizando computadoras.

Además, en muchas ocasiones el contexto impone restricciones que son importantes para la elaboración de la solución. Por ejemplo, si se está estudiando el software de un cajero automático, tener la respuesta a una consulta de saldo en menos de 10 minutos es un dato de contexto determinante sobre el tipo de solución aceptable.

La caracterización del contexto es muy amplia. Por ejemplo, el número de empleados que pueden ser atendidos simultáneamente, los clientes con diferente prioridad, la posibilidad de pagos con tarjetas de crédito, el tipo y la cantidad de líneas telefónicas disponibles, etc., son datos que acompañan al modelo esencial del sistema y deben tenerse en cuenta antes de diseñar una solución.

1.3.2 Descomposición de problemas. Concepto de modularización

En los párrafos anteriores se habló del pasaje del modelo general del problema a una descomposición en subproblemas menores (módulos), que en lo posible deben tener una función bien definida en el sistema.

La modularización o descomposición del modelo es muy importante y no solo se refiere a los procesos o funciones a cumplir, sino también a la distribución y utilización de los datos de entrada y salida, así como de los datos intermedios necesarios para alcanzar una solución.

Se verá más adelante que un módulo de programa encapsula una función y datos. Normalmente los datos que son propios del módulo serán llamados **locales** y aquellos datos que deba recibir o enviar a otros módulos serán datos de comunicación. Los datos que pueden ser conocidos y utilizados por todos los módulos de un sistema se llaman **globales**.

1.3.3 Concepto de algoritmo

Se mencionó anteriormente la palabra algoritmo, de la que seguramente el lector tiene al menos una idea imprecisa.

Definición

Algoritmo es la especificación rigurosa de la secuencia de pasos (instrucciones) a realizar sobre un autómata para alcanzar un resultado deseado en un tiempo finito.

En la definición precedente se utilizó la expresión especificación rigurosa a fin de significar que se debe expresar un algoritmo en forma unívoca, es decir, que la sintaxis y la semántica de cada instrucción deben estar perfectamente definidas.

Se utiliza el término autómata con un criterio generalizador de la palabra computadora. Se pueden ejecutar simbólicamente algoritmos sobre cualquier autómata, sea o no una computadora electrónica digital. De todos modos, para el alcance de este texto aceptaremos una equivalencia entre autómata y computadora.

Alcanzar el resultado en tiempo finito significa que se supone que un algoritmo comienza y termina. Una generalización de esta definición (por ejemplo, para sistemas de tiempo real que no terminan nunca, tales como el sistema de alarma mencionado precedentemente) cambia la terminología "tiempo finito" por "número finito" de instrucciones. De todos modos, en este texto se mantendrá la noción de principio y fin de los algoritmos, para simplificar la demostración de su corrección.

1.3.4 Las partes de un programa

Se ha discutido en los párrafos precedentes los componentes básicos de un programa: instrucciones y datos.

Las instrucciones (que también se han denominado acciones) representan las operaciones que ejecutará la computadora al interpretar el programa. Como se vio en los ejemplos anteriores, (y tal como se analizará en capítulos posteriores), todos los lenguajes de programación tienen un conjunto mínimo de instrucciones que deben contener asignación, decisión e iteración.

Puede demostrarse (Linger et al,1980) que un lenguaje con solo tres instrucciones: asignación, decisión e iteración, permite escribir cualquier algoritmo.

Los datos son los valores de información de los que se necesita disponer, y en ocasiones transformar, para ejecutar la función del programa.

Conceptualmente, por una parte se tienen datos constantes que no cambian durante la ejecución del programa. Tal es el caso del valor de la fuerza de gravedad en los ejemplos de Física o el valor de la Avenida de destino (avenida 50) en el ejemplo 1.2.

Por otra parte, se pueden tener datos variables, es decir, que durante la ejecución del programa pueden cambiar. Por ejemplo, el contenido de la bolsa del robot o los objetos depositados en cada esquina de la ciudad evolucionan dinámicamente con la ejecución del ejemplo anterior.

Tanto los datos constantes como los variables deben guardarse en la memoria de datos de una computadora. En ambos casos estarán representados simbólicamente por un nombre que se asocia con una dirección única de memoria. Por esto, el contenido de la dirección de memoria correspondiente a un dato constante se asigna una sola vez en los programas. En cambio, el contenido (valor) de la dirección de memoria correspondiente a un dato variable puede asignarse y cambiar muchas veces durante la ejecución del programa.

Además de datos e instrucciones, al leer un programa encontraremos comentarios, es decir, texto aclaratorio para el programador o el usuario, que no es entendido ni ejecutado por la computadora. Este texto sirve para clarificar qué hace el programa, y será de gran importancia cuando se intente modificarlo o corregirlo.

Una visión interesante de un programa es que corresponde a una transformación de datos. A partir de un contexto determinado por las precondiciones, el programa transforma la información y debería llegar al resultado esperado produciendo un nuevo contexto, caracterizado por las poscondiciones.

1.4 Software

La definición de Informática dada en el inicio de este capítulo, abarca aspectos relacionados tanto con la estructura propia de las computadoras como con su programación y control lógico.

Normalmente, los aspectos vinculados con la arquitectura física de las computadoras, las comunicaciones entre computadoras y los dispositivos periféricos que utilizan se denominan hardware.

A su vez, todo lo que se refiere a la programación básica y de aplicaciones, de modo de dar una utilidad definida al hardware, se denomina software.

Es interesante notar que a veces resulta muy difícil separar hardware y software. Por ejemplo, el controlador de un lavarropas programable es una pequeña computadora, que contiene hardware y software (los programas predefinidos que el usuario selecciona).

Esto explica la fuerte vinculación que hoy tienen la electrónica y la informática. Esencialmente la electrónica (hardware) provee el cambio tecnológico que impulsa el desarrollo de la informática pero, a su vez, los cambios de software (nuevos lenguajes, nuevos sistemas operativos, nuevos algoritmos) impactan en la concepción de circuitos y sistemas electrónicos.

Nuestro interés en el texto estará en el software y particularmente en el análisis y diseño de algoritmos dedicados a resolver problemas simples. Para ello, se estudiarán las clases de estructuras de datos adecuadas para resolver los problemas.

De todos modos, a fin de dar un marco al desarrollo y prueba de algoritmos, dentro de la temática general del software y en particular de la Ingeniería de Software (Pressman, 1993; Gehani y McGettrick, 1986) conviene repasar ordenadamente las etapas del desarrollo de una solución a un problema del mundo real que fue discutido a lo largo de este capítulo.

1.4.1 Etapas en la resolución de problemas con computadora

Análisis del problema.

En esta primera etapa, se analiza el problema en su contexto del mundo real. Deben obtenerse los requerimientos del usuario. El resultado de este análisis es un modelo preciso del ambiente del problema y del objetivo a resolver. Un componente importante de este modelo son los datos a utilizar y las transformaciones de los mismos que llevan al objetivo.

Diseño de una solución.

Suponiendo que el problema es computable, a partir del modelo se debe definir una estructura de sistema de hardware y software que lo resuelva. En este texto, se hablará solo del sistema de software. El primer paso en el diseño de la solución es la modularización del problema, es decir, la descomposición del mismo en partes que tendrán una función bien definida y datos propios (locales). A su vez, debe establecerse la comunicación entre los módulos del sistema de software propuesto.

Especificación de algoritmos.

Cada uno de los módulos del sistema de software tiene una función que se puede traducir en un algoritmo. La elección del algoritmo adecuado para la función del módulo es muy importante para la eficiencia posterior del sistema de software. Como se verá a lo largo del texto, una misma función sobre los mismos datos puede resolverse con una utilización de recursos muy diferente (memoria, tiempo) según el algoritmo elegido.

Escritura de programas.

Un algoritmo es una especificación simbólica que debe convertirse en un programa real sobre un lenguaje de programación concreto. Este proceso de programación tiende a automatizarse en la medida que los lenguajes algorítmicos se acercan a los lenguajes reales de programación. A su vez, el programa escrito en un lenguaje de programación determinado (por ejemplo, Pascal, Ada, C, Java) debe traducirse (automáticamente) al lenguaje de máquina de la computadora que lo vaya a ejecutar. Esta traducción, denominada compilación, permite detectar y corregir los errores sintácticos que se cometan en la escritura del programa.

Verificación.

Una vez que se tiene un programa escrito en un lenguaje real y depurado de errores sintácticos, se debe verificar que su ejecución conduzca al resultado deseado, con datos representativos del problema real. Sería deseable poder afirmar que el programa es correcto, más allá de

los datos particulares de una ejecución. Sin embargo, en los casos reales es muy difícil realizar una verificación exhaustiva de todas las posibles condiciones de ejecución de un sistema de software. La facilidad de verificación y la depuración de errores de funcionamiento del programa conducen a una mejor calidad del sistema y este es el objetivo central de la Ingeniería de Software (Gehani, McGetrick: Software Specification Techniques - International Computer Science Series, 1986).

Nótese que un error de funcionamiento de alguno de los programas del sistema puede llevar a la necesidad de rehacer cualquiera de las etapas anteriores, incluso volver a discutir los requerimientos.

A lo largo del texto se discutirán problemas en los que se ha simplificado la complejidad del mundo real, a fin de tenerlos prácticamente analizados.

Se pondrá especial énfasis en el diseño e implementación de soluciones, entendiendo por tales la modularización, la elección de las estructuras de datos adecuadas y la especificación de algoritmos que se conviertan en programas concretos verificables. Técnicamente es lo que se denomina *programming in the small.*

Los algoritmos que se analizarán serán básicamente orientados a control, lo que normalmente se denomina programación imperativa, y el lenguaje básico de ejemplificación será Pascal.

El capítulo 12 abre la discusión sobre paradigmas de programación alternativos a la programación imperativa, tales como la programación funcional, la programación lógica y la orientación a objetos. No se discute en el texto la posibilidad de descomponer algoritmos en procesos que se puedan ejecutar concurrentemente sobre diferentes procesadores (Andrews, 1991). Nuestro modelo de computadora se corresponderá con un procesador único que sigue el esquema de la Figura 1.1 y que es controlado por una secuencia de instrucciones de acuerdo al funcionamiento explicado en la sección 1.1.

Conclusiones

Se han presentado algunas definiciones básicas, que sirven como conceptos introductorios al resto del texto.

El lector deberá asociar inmediatamente el proceso de desarrollo de software con el proceso de refinamiento y abstracción que abarca desde el problema real hasta su solución algorítmica con un lenguaje de programación.

En las referencias bibliográficas se presenta una visión ampliada de los conceptos presentados en este capítulo.

2

Capítulo 2

Algoritmos

2

Algoritmos

Objetivo

En este capítulo se parte de la definición de algoritmo como especificación ordenada de la solución a un problema del mundo real.

El eje temático está centrado en las estructuras de control que permitirán expresar la secuencia de decisiones y de órdenes a ejecutar por una computadora en la solución de un problema.

Asimismo, se introducen los conceptos de modularización o descomposición de un problema como un mecanismo de reducción de la complejidad y simplificación de la tarea a resolver; de documentación de algoritmos como un requerimiento para permitir soluciones que se puedan comprender y mantener en el tiempo; de eficiencia como una medida de la *performance* del algoritmo elegido y de corrección de los algoritmos en la búsqueda de soluciones que respondan a los requerimientos del mundo real.

Por último, partiendo del concepto de datos simples y de variables vistos en el primer capítulo, se trabaja sobre la noción de localidad de las variables en un programa o módulo de programa.

2.1 Estructuras de control

En el capítulo 1 se definió algoritmo como la especificación rigurosa de la secuencia de pasos (instrucciones) a realizar sobre un autómata para alcanzar un resultado deseado en un tiempo finito.

Las instrucciones representan las operaciones que ejecutará la computadora al implementar el algoritmo. Estas instrucciones utilizan cierta información en forma de constantes o variables, que se denominan datos.

Todos los lenguajes de programación tienen un conjunto mínimo de instrucciones que permiten especificar el control propio del algoritmo que se requiere implementar. Este conjunto mínimo debe contener estructuras para la asignación, decisión e iteracción.

2.1.1 Estructuras de control básicas

2.1.1.1 Secuencia

La estructura de control más simple está representada por una sucesión de operaciones (por ej. asignaciones, donde asignar significa almacenar valores constantes o variables), en la que el orden de ejecución coincide con el orden físico de aparición de las instrucciones.

La Fig. 2.1 muestra el esquema de una secuencia (por lo que se denominan estructuras secuenciales), que en algunos lenguajes de programación se suele encerrar entre las palabras claves `begin` y `end` para indicar el comienzo y fin de la misma.

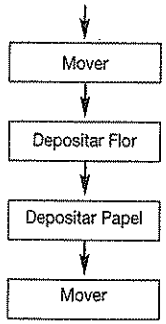


Figura 2.1

Por ejemplo, un recorrido en el cual el robot del Visual Da Vinci comienza desde la posición (1,1) con flores y papeles en su bolsa, da un paso, deposita una flor, deposita un papel, da otro paso y luego se detiene, se puede realizar mediante una secuencia como la siguiente:

Ejemplo 2.1

Precondiciones

- El robot está ubicado en la esquina (1,1)
- En su bolsa tiene flores y papeles

Poscondiciones

- El robot se encuentra en la esquina (1,3)
- En su bolsa hay una flor y un papel menos

```
{ El robot está ubicado en (1,1) y en su  }
{ bolsa tiene flores y papeles            }

programa Uno
comenzar
  iniciar
  mover
  depositarFlor
  depositarPapel
  mover
fin

{ El robot se encuentra en (1,3) y        }
{ en su bolsa hay una flor y un papel menos }
```

2.1.1.2 Repetición

Una extensión natural de la secuencia consiste en repetir n veces un bloque de acciones.

Se considera que el número de veces que se deben ejecutar las acciones es fijo y conocido de antemano.

La Fig. 2.2 muestra el diagrama esquemático de la repetición.

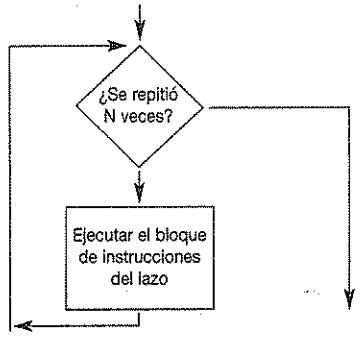


Figura 2.2

En el ejemplo anterior, si el robot debe repetir 10 veces las acciones de: avanzar, depositar una flor y depositar un papel (suponiendo que tiene suficientes papeles y flores en su bolsa), a partir de (1,1) resultaría el código del ejemplo 2.2:

Ejemplo 2.2

Precondiciones

- El robot tiene al menos 10 flores y 10 papeles en su bolsa
- Está ubicado en (1,1)

Poscondiciones

- En la bolsa ahora tiene 10 flores y 10 papeles menos
- Está ubicado en la esquina (1,11)

```
{ El robot tiene al menos 10 flores y 10 papeles en sus   }
{ bolsas y está ubicado en (1,1)                          }

programa Dos
comenzar
   iniciar
   repetir 10
      mover
      depositarFlor
      depositarPapel
fin

{ En las bolsas ahora tiene 10 flores y 10 papeles menos, }
{ y está ubicado en (1, 11)                               }
```

Para esta estructura la sintaxis comúnmente utilizada por los lenguajes de programación es la siguiente:

Código

```
For indice := valor_inicial to valor_final do
    Acciones
End do
```

En el ejemplo 2.3 se muestra su implementación, considerando que la variable A es de tipo entera.

Ejemplo 2.3

```
{ Contador es una variable de tipo INTEGER }

A = 100;
for Contador := 1 to 10 do
   A := A + 2;
Imprimir(A);

{ A contiene el valor 120 }
```

Algunas consideraciones generales de la estructura de control `For` son:

- `For`, `to`, `do` y `End do` son palabras clave del lenguaje. Hay que definir contador como una variable que se incrementa en una cantidad constante.
- `Indice` es una variable de tipo ordinal (tal como `contador` en el ejemplo anterior) que permite controlar el número de ejecuciones del ciclo.
- `Valor_Inicial` y `Valor_Final` son valores que indican el o los límites entre los que varía índice al comienzo y final del ciclo. Se debe notar que `Valor_Final - Valor_Inicial + 1` indica el número de veces que se ejecuta el ciclo.
- Está implícito que en cada ciclo la variable índice toma el valor siguiente de acuerdo al tipo ordinal asociado a dicha variable. En el ejemplo, el **incremento es 1**. Se verá que en los diferentes lenguajes de programación se puede modificar este incremento o hacerlo negativo. De todos modos, conceptualmente la variable `índice` se actualiza en cada ciclo y se testea respecto al valor final deseado.
- Dentro de las Acciones del ciclo **no es posible modificar** la variable `índice`. Al terminar el ciclo, la variable `índice` no tiene un valor definido (su utilidad se limita a la estructura de repetición).

Otros ejemplos:

Ejemplo 2.4

```
{ Este bloque se ejecuta 8 veces }
{ Se supone Indice como una variable CHAR }

for Indice := 'A' to 'H' do
begin
   Acciones;
end;
```

Ejemplo 2.5

```
{ Este bloque se ejecuta 2 veces              }
{ Se supone Indice como una variable BOOLEAN }

for Indice := FALSE to TRUE  do
begin
  Acciones;
end;
```

Ejemplo 2.6

```
{ Este bloque NO se ejecuta                   }
{ Se supone Indice como una variable INTEGER }

for Indice := 20 to 18 do
begin
  Acciones;
end;
```

2.1.1.3 Decisión

En un algoritmo representativo de un problema real es prácticamente imposible que las instrucciones sean secuenciales puras. Es necesario tomar decisiones en función de los datos del problema.

La estructura básica de decisión entre dos alternativas es la que se representa simbólicamente en la Fig.2.3.

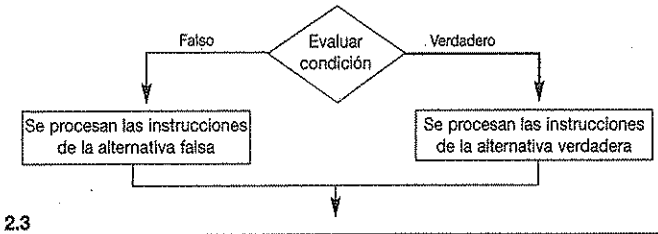


Figura 2.3

La sintaxis común para esta estructura en los lenguajes de programación es la siguiente:

Código

```
If (condición) then
                Acciones_por_Condición_Verdadera ;
Else
                Acciones_por_Condicion_Falsa ;
End if
```

- `If`, `then`, `else` y `End if` son palabras clave del lenguaje.

- Condición es una expresión que al evaluarse devuelve un valor lógico `TRUE` o `FALSE` para tomar la decisión.

Por ejemplo:

Se tienen 3 variables enteras `A`, `B` y `Min` y se desea asignar en la variable Min el menor de los valores entre A y B. En el Ejemplo 2.7 se puede ver su implementación.

Ejemplo 2.7

```
{ A, B, Min son variables INTEGER       }

if A <= B then
  Min := A
else
  Min := B;

{ Min tiene el menor valor entre A y B }
```

Utilizando el ejemplo 2.2, supóngase ahora que el robot debe recoger un papel en cada calle impar a partir de (1,3); en caso de no poder hacerlo debe depositar otra flor. El segmento de algoritmo del Ejemplo 2.8 presenta los cambios necesarios.

Ejemplo 2.8

Precondiciones

- **El robot está en la esquina (1,1)**
- **El robot cuenta con la cantidad de flores necesarias para cumplir el ciclo.**

Poscondición

- **Todas las esquinas con calles pares tienen flores y papeles.**

```
{ El robot está en (1,1). Cuenta con la cantidad de flores   }
{ necesarias para cumplir el ciclo                           }

programa Tres
comenzar
  iniciar
  repetir 10
    mover
    depositarFlor
    depositarPapel
    mover
    si HayPapelEnLaEsquina
      tomarPapel
    sino
      depositarFlor
fin
{ Todas las esquinas con calles pares tiene flores y papeles }
```

Puede ocurrir que no se tengan que representar acciones cuando la condición es falsa. Entonces la estructura quedaría reducida a:

Código

```
IF (condición)  Then
                Acciones_por_Condición_Verdadera
End If
```

Utilizando el ejemplo 2.7, el algoritmo puede reescribirse del siguiente modo:

Ejemplo 2.9

```
{ A, B, Min variables INTEGER }

Min := B;
if A <= B then
   Min := A;

{ Min tiene el menor valor entre A y B }
```

Los lenguajes de programación pueden tener variantes que enriquezcan la estructura de decisión que se ha presentado (por ejemplo en ADA se tiene una variante de la estructura de control el `IF OR ELSE IF` e `IF AND THEN IF`) pero, normalmente, todos los lenguajes disponen de la estructura originalmente definida.

2.1.1.4 Selección

La Fig. 2.4 muestra una extensión de la estructura básica de decisión, para el caso de que las alternativas sean más de dos:

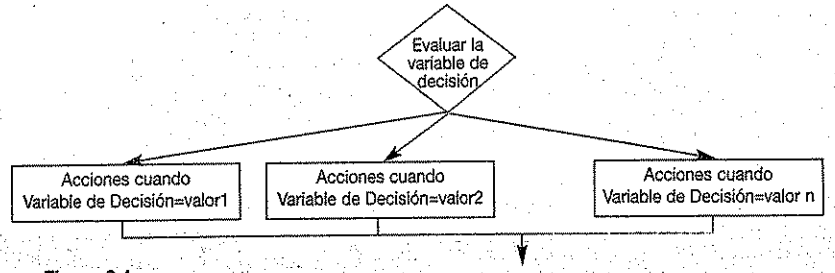


Figura 2.4

La sintaxis común en los lenguajes de programación es:

Código

```
Select Case ( Variable_Decisión )
                    Case (posibilidad 1)
                          Acciones para posibilidad 1 Verdadera ;
                    Case (posibilidad 2)
                          Acciones para posibilidad 2 Verdadera ;
                          .............
                    Case (posibilidad N)
                          Acciones para posibilidad N Verdadera ;
                    Else
                          Acciones para todas las otras posibilidades
                          no contempladas arriba;
End Select
```

Algunas consideraciones generales de la estructura de control `Case` son:

- `Select Case`, `Case`, `Else` y `End Select` son palabras clave del lenguaje.
- Posibilidad n es una expresión que involucra a la `Variable_Decisión`. Al evaluarse devuelve un valor lógico, si es `TRUE` se ejecutan las acciones para esa posibilidad.
- Algunos lenguajes imponen que alguna de las n condiciones debe ser verdadera. En estos casos no existe cláusula `Else`.
- Los conjuntos de valores permitidos para cada posibilidad son disjuntos.

Ejemplo 2.10

El siguiente segmento de algoritmo permite leer las edades de personas entre 1 y 100 años. Se desea agrupar a las edades de las personas en 4 categorías: menores de 10 años, entre 10 y 25 años, entre 26 y 45 años y entre 46 y 100 años.

Precondición

- **Se analiza un grupo de personas cuyas edades oscilan entre 1 y 100 años.**

Poscondición

- **En cada uno de los casos se tiene la cantidad total de personas que pertenecen a los intervalos correspondientes**

```
{ edad es una variable entera positiva}
{ clase1, clase2, clase3 y clase4 con contadores enteros}
{ dato invalido es un contador para edades fuera de rango}
{ inicializar contadores en 0
{ leer la variable edad}
```

```
Select Case (edad)
                    Case 1..9
                       Clase1 :=clase1+1;
                    Case 10..25
                       Clase2 :=clase2+1;
                    Case 26..45
                       Clase3 :=clase3+1;
                    Case 46..100
                       Clase4 :=clase4+1;
                    Else
                       Dato_invalido :=dato_invalido + 1;
End Select
```

Ejemplo 2.11

Se supone que el robot está ubicado en una esquina cualquiera, y debe resolver qué dirección debe tomar de acuerdo a la cantidad de flores que tenga en la bolsa. Para ello debe averiguar, si hay 5 flores debe girar a la derecha, si hay 3 flores debe girar a la izquierda, si hay solo una flor debe seguir en la misma dirección, y en caso contrario debe quedarse en el mismo lugar.

Precondiciones
- El robot está ubicado en (1,5)
- De acuerdo a la cantidad de flores puede tomar una de las acciones.

Poscondición
- El robot queda orientado como corresponde

```
{ El robot está ubicado en (1,5) }

programa Cuatro
variables
  Flor: numero
comenzar
  iniciar
  pos (1,5)
  Flor := 0
  mientras HayFlorEnLaBolsa
     depositarFlor
     Flor := Flor + 1
  caso (Flor)
     5: Derecha

     3: repetir 3
          derecha

     1: Mover
fin
```

Nota: En la versión 1.0 del ambiente Visual Da Vinci la estructura caso no está implementada.

2.1.1.5 Iteración

Puede ocurrir que se desee ejecutar un bloque de instrucciones desconociendo el número exacto de veces que se ejecutan. Para estos casos en la mayoría de los lenguajes de programación estructurada existen **estructuras de control iterativas condicionales**. Como su nombre lo indica, las acciones se ejecutan dependiendo de la evaluación de la condición definitiva.

Estas estructuras se clasifican en **pre-condicionales** y **pos-condicionales**. Las pre-condicionales evalúan la condición y, si es verdadera, se ejecuta el bloque de acciones; esto hace que dicho bloque se pueda ejecutar 0, 1 o más veces.

La Fig.2.5 muestra el esquema correspondiente:

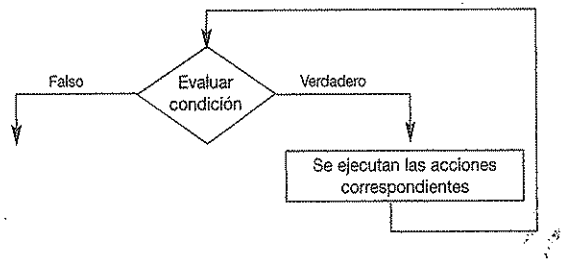


Figura 2.5

Ejemplo 2.12

Se desea resolver un algoritmo que lea una sucesión de números enteros, la cual finaliza al ingresar el valor 0. Se desea obtener la suma de los números leídos.

Precondición
- Se lee una secuencia de números que finaliza en 0.

Poscondición
- Se obtiene la suma total de los números.

```
{ En la variable Suma se guarda el resultado de la   }
{ suma de los números ingresados.                    }
{ Se debe inicializar la variable Suma               }

Suma := 0;
WriteLn('Ingrese un número entero: ');
{ La variable Numero toma el valor que luego se utiliza para evaluar
  la condición por primera vez  }
Read(Numero);
while Numero <> 0 do
```

```
begin
  Suma := Suma + Numero;
  WriteLn('Ingrese un número entero: ');
  { La variable Numero toma otro valor que utilizará como nueva
    condición }
  Read(Numero);
end;
WriteLn('El resultado de la suma es: ', Suma);
```

Ejemplo 2.13

Supóngase que el robot debe recoger todos los papeles de la esquina donde se encuentre ubicado, la solución podría escribirse como sigue:

Precondición

- El robot se encuentra ubicado en la esquina (10,10)

Poscondición

- No hay papeles en la esquina (10, 10)

```
{ El robot se ubica en la esquina (10,10) }

programa Cinco
comenzar
  iniciar
  Pos(10, 10)
  mientras HayPapelEnLaEsquina
     tomarPapel
fin

{ No hay papeles en la esquina (10,10)    }
```

Este ciclo termina cuando en la esquina no hay más papeles

En cuanto a las estructuras poscondicionales, primero se ejecuta el bloque de acciones, luego se evalúa la condición y, si es falsa, se ejecuta nuevamente el bloque de acciones. A diferencia de la estructura iterativa anterior, el bloque de acciones se debe ejecutar 1 o más veces. Nótese que, en este caso, el bloque de acción se ejecuta antes de evaluar la condición, por lo tanto se lleva a cabo al menos una vez. La Fig 2.6 muestra el esquema correspondiente.

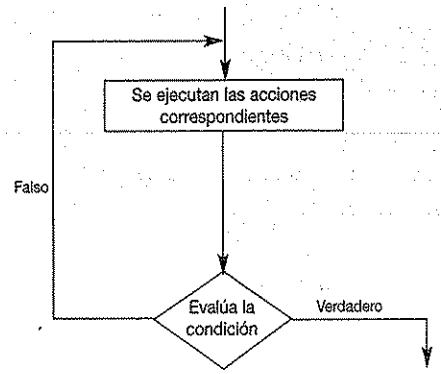


Figura 2.6

Ejemplo 2.14

Supóngase que se desea procesar una secuencia de números para obtener su suma. Dicha secuencia termina cuando se ingresa el número 23, el que también debe ser procesado.

Precondición

- El primer número es distinto de 0

Poscondición

- Se obtiene la suma de los números ingresados

```
{ En la variable Suma se guarda el resultado de la }
{ suma de los números ingresados                   }
{ Se debe inicializar la variable Suma en 0        }

Suma := 0;
repeat
  WriteLn('Ingrese un número entero: ');
  Read(Numero);
  Suma := Suma + Numero;
until Numero = 0;
WriteLn('El resultado de la suma es: ', Suma);

{ En Suma se acumula la suma de los números leídos }
```

Ejemplo 2.15

Si el robot debe recorrer la avenida 1 hasta encontrar una esquina que tiene exactamente 5 flores, depositando todo el contenido de la bolsa en la esquina siguiente del recorrido, se podría escribir la siguiente solución:

Precondiciones

- **El robot está ubicado en la esquina (1,1).**
- **En la avenida 1 existe una esquina con 5 flores.**

Poscondición

- **El robot encontró una esquina con 5 flores.**

```
{ El robot está ubicado en la (1,1).                    }
{ La esquina seguro existe y es anterior a la calle 100 }

programa Seis
variables
  Flores: numero
comenzar
  iniciar
  mientras Flores <> 5
    Flores := 0
    mientras HayFlorEnLaEsquina
      Flores := Flores + 1
      tomarFlor
    mover
  mientras HayFlorEnLaBolsa
    depositarFlor
fin
{ El robot encontró una esquina con 5 flores            }
```

2.1.2 Estructura esquemática de un programa

A continuación se presenta un caso que plantea la estructura esquemática de un programa, siguiendo para ello la sintaxis del lenguaje Pascal.

Código

```
Program Identificador;
{ sección de declaración de datos}
    Var  id1: tipo;
         id2: tipo;
         .............
   { sección de sentencias ejecutables}
     begin

     end.
```


Donde `Program`, `Var`, `Begin` y `End` son palabras claves:

- `Program` indica el comienzo de un programa.
- `Identificador` es el nombre que se le dio al programa.
- `Var` indica la sección de declaración de los datos variables a utilizar en el programa. Esta sección se extiende hasta la palabra `begin`.
- `Begin` indica el comienzo de la sección de sentencias ejecutables del programa. Esta sección se extiende hasta la palabra `end`.
- `End.` indica el fin del programa.
- Los comentarios se escriben entre llaves en cualquier parte del programa.

Ejemplo 2.16

El siguiente algoritmo permite calcular la superficie de un terreno a partir de sus dimensiones.

Precondición

- **Las longitudes del terreno son distintas de 0**

Poscondición

- **La variable superficie contiene el valor del área del terreno**

```
{ Este programa calcula la superficie de un terreno }
{ Las longitudes del terreno son distintas de 0     }

program Superficie;
{ Sección de declaración de variables }
var
  Lado1: integer;       { longitud de un lado del terreno    }
  Lado2: integer;       { longitud de otro lado del terreno  }
  Superficie: integer;  { representa el área del terreno     }
begin
  { Sección de instrucciones ejecutables }
  WriteLn('Ingrese la longitud de un lado: ');
  ReadLn(Lado1);
  WriteLn('Ingrese la longitud de otro lado: ');
  ReadLn(Lado2);
  { Cálculo de la superficie }
  Superficie := Lado1 * Lado2;
  { Salida del resultado }
  WriteLn('La superficie del terreno es: ', Superficie);
end.
```

2.2 Importancia de la documentación de un algoritmo

Un programa bien documentado será más fácil de leer y mantener. Para ello la documentación es fundamental. Es común que la mayoría de los lenguajes de programación provean algún mecanismo de documentación, por ejemplo, a través de la inserción de comentarios en el programa. Un programa sin comentarios revela un estilo de programación pobre y peligroso, ya que dificulta el mantenimiento adecuado del mismo.

Es recomendable utilizar un comentario general para el objetivo del programa o del módulo de programa en cuestión, que refleje la especificación del problema a resolver, de la forma más completa posible. Su lectura debe ser suficiente para poder entender las acciones que se llevan a cabo en el mismo. Además, deberán describirse todos los datos variables y constantes que intervengan en el programa, e indicar la fecha de realización del mismo, el autor, etc.

Por otra parte, se deberán elegir identificadores de manera tal que sean autoexplicativos. En cuanto a los comentarios intercalados en el programa (documentación on-line o interna), deben realizarse con cierto criterio, pues deben contribuir a la claridad del programa; por ejemplo, es importante comentar el estado inicial y final entre las acciones de una determinada secuencia, los efectos colaterales probables que puedan tener lugar luego de una llamada a un módulo, etc.

Es importante destacar que cuando se realiza el mantenimiento de un programa no solo se actualiza el código, sino también los comentarios del mismo. Debe quedar claro que los comentarios no son un agregado al programa, sino una parte importante del mismo.

2.3 Corrección de algoritmos. Importancia de la verificación

Un programa es correcto cuando cumple con su especificación, esto significa que cumple con los requerimientos propuestos. Para determinar cuáles son esos requerimientos se debe tener una especificación completa, precisa y libre de ambigüedades del problema a resolver antes de escribir el mismo.

Una de las formas para saber si un programa es correcto consiste en probar con un juego de datos reales que permitan determinar que cumple con su especificación. Debido a la importancia de la verificación de los programas las técnicas empleadas evolucionan en búsqueda de mayor eficiencia. Este campo de investigación se actualiza constantemente en la Ciencia de la Computación.

Ejemplo 2.17

Supóngase que el robot debe vaciar su bolsa de flores depositando una flor en cada una de las esquinas de la ciudad. A continuación se presenta la solución a dicho problema.

Precondición

- **El robot está ubicado en la esquina (1,1)**

```
{ El robot está ubicado en la esquina (1,1) }

programa Siete
comenzar
  iniciar
  mientras HayFlorEnLaBolsa
    depositarFlor
    mover
fin
```

¿Pero esta solución es siempre correcta? La respuesta es **no.**

Y una de las razones surge ante la pregunta: ¿qué ocurre si en la bolsa hay más de 100 flores?

Una solución correcta, utilizando las mismas precondiciones, sería:

Ejemplo 2.18

```
programa Ocho
variables
  Pasos: numero
comenzar
  iniciar
  mientras HayFlorEnLaBolsa
    Pasos := 0
    mientras HayFlorEnLaBolsa & (Pasos < 99)
      depositarFlor
      mover
      Pasos := Pasos + 1
    si Pasos = 99
      derecha
      mover
      derecha
  Informar(PosAv, PosCa)
fin
```

2.4 Eficiencia de un algoritmo

A lo largo del texto se verá que se pueden tener varias soluciones algorítmicas para un mismo problema (por ejemplo, para encontrar el menor valor de una lista de números o para que el robot vaya de un punto a otro de la ciudad). Sin embargo, el uso de recursos (tiempo, memoria) para cada una de esas soluciones puede ser muy diferente.

Definición

Se define a la eficiencia como una métrica de calidad de los algoritmos, asociada con una utilización óptima de los recursos del sistema de cómputo donde se ejecutará el algoritmo.

Ejemplo 2.19

Supóngase que el robot debe ir desde la esquina (1,1) a la esquina (21,1). Un recorrido posible consiste en recorrer la calle 1 desde la avenida 1 a la avenida 21. Para este recorrido, la solución sería:

Precondición

- El robot está ubicado en la esquina (1,1)

Postcondiciones

- El robot está ubicado en la esquina (21,1)
- Caminó 20 cuadras

```
{ El robot está ubicado en la esquina (1,1)  }

programa Nueve
comenzar
  iniciar
  derecha
  repetir 20
    mover
fin

{ El robot está ubicado en la esquina (21,1) }
{ El robot caminó 20 cuadras                 }
```

Otro recorrido posible es el que se muestra en la Figura 2.7

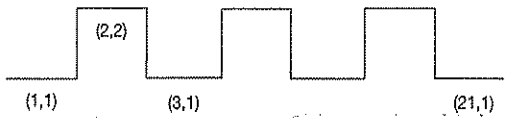


Figura 2.7

Ejemplo 2.20

Precondición

- El robot está ubicado en la esquina (1,1)

Poscondición

- El robot está ubicado en la esquina (21,1) y caminó 40 cuadras

```
{ El robot está ubicado en la esquina (1,1)  }

programa Diez
comenzar
  iniciar
  derecha
  repetir 10
    mover
    repetir 3
      derecha
    repetir 2
      mover
      derecha
    mover
    repetir 3
      derecha
fin

{ El robot está ubicado en la esquina (21,1) }
{ El robot caminó 40 cuadras                 }
```

Obsérvese que en ambas soluciones el robot llega a la esquina (21,1). En la primera solución camina 20 cuadras y en la segunda solución camina 40 cuadras.

2.5 Descomposición de problemas

2.5.1 Modularización

Hasta aquí se han analizado los algoritmos y los programas que dan solución a problemas simples y breves. Sin embargo, se podrá observar que, en general, los problemas del mundo real son más complejos y extensos. En algunas ocasiones, por ejemplo, una solución ya implementada deberá modificarse para satisfacer un nuevo requerimiento. Es por ello que se deben considerar algunas herramientas para facilitar la resolución de tales problemas. Algunas de estas herramientas tienen que ver con la abstracción y con la descomposición de los problemas a resolver. La abstracción permitirá representar los objetos relevantes en el dominio del problema,

y la descomposición es una técnica que se basa en el paradigma: "Divide y vencerás". La descomposición tiene un objetivo: dividir cada problema en subproblemas y estos, a su vez, en subproblemas más pequeños y así sucesivamente. Cada uno de los subproblemas finales resultará entonces más simple de resolver.

Cuando se realiza esta descomposición debe tenerse en cuenta que cada subproblema está en un mismo nivel de detalle, que cada uno se puede resolver independientemente de los demás y que las soluciones de estos subproblemas pueden combinarse para resolver el problema original.

El método de diseño descendente (*Top Down*) se lo conoce también como modularización y es importante para lograr un buen diseño de los programas. Esquemáticamente se la representa en la Figura 2.8

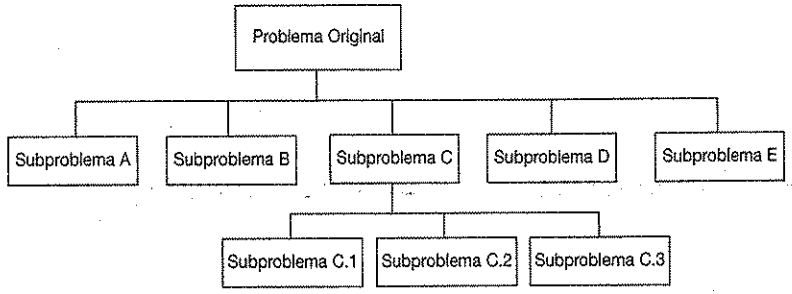


Figura 2.8

2.5.1.1 Importancia de la modularización

¿Por qué es importante una buena modularización?

El Conocimiento humano.

Algunas investigaciones acerca de la cognición han demostrado que existe un límite en cuanto a la cantidad de información que una persona puede tener en su mente a la vez, y que esa cantidad es de 7 +/- 2 piezas de información. También demostraron que naturalmente la persona tiende a entender un problema de características complejas "partiendo" o "trozando" la información. Es decir, mientras la persona puede tener 7 u 8 piezas de información en mente a la vez, cada una de estas piezas, puede ser parte de otras 7 u 8 piezas distintas, estableciéndose de esta manera una relación jerárquica. En consecuencia, para comprender un problema complejo del mundo real es necesario dividir y subdividir, o sea: modularizar.

Trabajo en equipo.

La modularización favorece el trabajo en equipo ya que los productos de software complejos necesitan más de una persona para poder realizar el diseño, la codificación y la corrección de los mismos.

Por ejemplo, un programador normalmente trabaja desarrollando una parte del código (algunos módulos) desconociendo el resto del trabajo.

Como el programa esta modularizado, cada programador recibe las especificaciones de la tarea a realizar y las restricciones con las que debe manejarse.

Mantenimiento del programa.

Esta actividad involucra básicamente dos tareas: corregir los errores y modificar el código. A menudo, los errores pueden ocurrir antes de que el producto de software esté terminado o bien pueden ser descubiertos más tarde, es decir, cuando el producto ya se está utilizando. De todos modos, estos errores deben ser identificados, ya que se deben encontrar las causas y el código correspondiente debe ser corregido. Como es usual que se realicen cambios al software, por ejemplo, para el agregado de nuevas funcionalidades, esto también lleva a modificar el código existente.

La experiencia demuestra que el costo de mantenimiento en un código que no ha sido apropiadamente modularizado es muy alto. En cambio, el costo puede reducirse si el código está bien modularizado dado que el tiempo requerido para entender el programa y luego modificarlo es notoriamente menor.

Otro problema común cuando se realiza el mantenimiento de un programa es tener en cuenta los *side effects* (efectos colaterales), los que aparecen al cambiar el código para satisfacer cierta necesidad y producen efectos no deseados en otras partes del programa. Las causas de estos efectos y cómo prevenirlos se verán más adelante.

Importancia de la reusabilidad del código.

La idea básica es muy simple: cada vez que se crea un nuevo programa es deseable, de ser posible, hacer uso del código que ya se ha escrito. Esto es posible si se complementan soluciones bien modularizadas.

2.5.1.2 Módulos de un programa

Un buen diseño de programa significa, entre otras consideraciones, que un problema sea descompuesto en un número de subproblemas, que se denominarán a partir de aquí módulos. Cada módulo tendrá, entonces, una tarea específica bien definida y se comunicarán entre sí adecuadamente para conseguir un objetivo común.

Un módulo es simplemente un "trozo de código", o un conjunto de instrucciones más un conjunto de definiciones de datos que realiza una tarea lógica.

En el robot, cada uno de estos módulos de programa se denomina proceso.

Volviendo al ejemplo del robot: se desea que el robot realice recorridos en escalera, recogiendo todos los papeles que encuentre. La primera escalera comienza en la esquina (1,1) y fi-

naliza en la esquina (10,10). La segunda escalera comienza en la esquina (3,1) y finaliza en la esquina (10,8). La tercera escalera comienza en la esquina (5,1) y finaliza en la esquina (10,6), y así sucesivamente. En todos los casos cada escalón debe ser de una cuadra, debe informar para cada recorrido en escalera la cantidad de pasos dados, y por último la cantidad de papeles recogidos en todo el recorrido.

Se debe notar que existe un conjunto de soluciones, donde solo la última será realmente operativa en el ambiente Visual Da Vinci. Desde un punto de vista didáctico se trata de mejorar el esquema del programa teniendo en cuenta aspectos de la modularización, y luego de variables locales y parámetros, hasta llegar a la solución deseada.

El algoritmo resultante se presenta así:

Ejemplo 2.21

Precondición

- El robot está en (1,1) orientado hacia el norte

Poscondiciones

- El robot queda posicionado en (10,2), orientado hacia el norte
- Cuenta la cantidad total de papeles en todo el recorrido

```
{ El robot está en (1,1) orientado hacia el norte }

programa Recorrido
variables
   Escalones: numero      { Indica la cantidad de escalones de cada
                            recorrido }
   Papeles: numero        { Indica la cantidad de papeles de cada
                            recorrido }
   TotalPapeles: numero   { Indica la cantidad total de papeles de
                            todos los recorridos }
   Avenida: numero        { Indica la avenida en la que comienza el
                            recorrido }
   Pasos: numero          { Indica los pasos dados en cada recorrido }
comenzar
   iniciar
   { Inicializar las variables (1)}
   Escalones := 9
   TotalPapeles := 0
   Avenida := 1
   { Hacer los 5 recorridos }
   repetir 5
      { Hacer una escalera contando la cantidad de papeles (2)}
      Papeles := 0
      Pasos := 0
```



```
      repetir Escalones
        mientras HayPapelEnLaEsquina
          tomarPapel
          Papeles := Papeles + 1
        mover
        Pasos := Pasos + 1
        derecha
        mientras HayPapelEnLaEsquina
          tomarPapel
          Papeles := Papeles + 1
        mover
        Pasos := Pasos + 1
        repetir 3
          derecha
        { Juntar los papeles de la última esquina del recorrido }
        mientras HayPapelEnLaEsquina
          tomarPapel
          Papeles := Papeles + 1
      { Informar la cantidad de pasos encontrados }
      Informar(Pasos)
      { Incrementar total de papeles del recorrido (3)}
      TotalPapeles := TotalPapeles + Papeles
      { Reposicionar al robot y actualizar cantidad de escalones (4)}
      Avenida := Avenida + 2
      Pos(Avenida, 1)
      Escalones := Escalones - 2
      { Informar la cantidad total de papeles del recorrido completo (5)}
      Informar(TotalPapeles)
fin

{ El robot queda posicionado en (10,2), mirando hacia el norte y en   }
{ TotalPapeles tiene la cantidad total de papeles de todo el recorrido }
```

Se puede observar que en este programa hay cinco módulos (numerados a continuación de los comentarios). El primero obtiene los datos iniciales para realizar el recorrido. Una vez obtenida esta información, el segundo realiza el recorrido en escalera, contando los papeles encontrados y la cantidad de pasos dados. Luego, se puede actualizar la cantidad total de papeles del recorrido. En el paso siguiente se posiciona al robot y se recalcula la cantidad de escalones para el nuevo recorrido y, por último, se informa la cantidad total de papeles encontrados.

Si bien es cierto que este programa resulta corto y simple de resolver de manera lineal, es aconsejable utilizar los beneficios de la modularización referenciados al principio del tema. Para reflejar una solución modularizada se proponen cinco procesos, uno para cada módulo de acuerdo a la Figura 2.9.

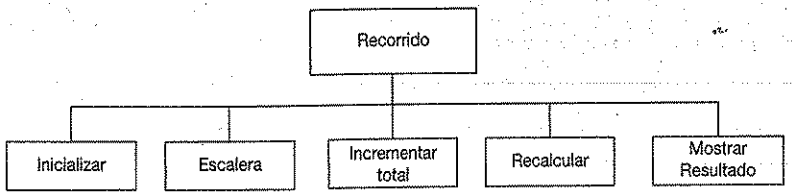


Figura 2.9

Se puede escribir, entonces, la siguiente solución:

Ejemplo 2.22

```
{ El robot está en (1,1) orientado hacia el norte }

programa Recorrido

variables
  Escalones: numero        { Indica la cantidad de escalones de cada
                             recorrido }
  Papeles: numero          { Indica la cantidad de papeles de cada
                             recorrido }
  TotalPapeles: numero     { Indica la cantidad total de papeles de
                             todos los recorridos }
  Avenida: numero          { Indica la avenida en la que comienza el
                             recorrido }
  Pasos: numero            { Indica los pasos dados en cada recorrido }

procesos
  proceso Inicializar
  comenzar
    Escalones := 9
    TotalPapeles := 0
    Avenida := 1
  fin

  proceso Izquierda
  comenzar
    repetir 3
      derecha
  fin
```



```
  proceso Escalera
  comenzar
    Papeles := 0
    Pasos := 0
    repetir Escalones
      mientras HayPapelEnLaEsquina
        tomarPapel
        Papeles := Papeles + 1
      mover
      Pasos := Pasos + 1
      derecha
      mientras HayPapelEnLaEsquina
        tomarPapel
        Papeles := Papeles + 1
      mover
      Pasos := Pasos + 1
      Izquierda
      { juntar los papeles de la última esquina del recorrido }
      mientras HayPapelEnLaEsquina
        tomarPapel
        Papeles := Papeles + 1
      { Informar la cantidad de pasos dados }
      Informar(Pasos)
  fin

  proceso IncrementarTotal
  comenzar
    TotalPapeles := TotalPapeles + Papeles
  fin

  proceso Recalcular
  comenzar
    Avenida := Avenida + 2
    Pos(Avenida, 1)
    Escalones := Escalones - 2
  fin

  proceso MostrarResultado
  comenzar
    Informar(TotalPapeles)
  fin

comenzar { programa principal }
  iniciar
  Inicializar
  repetir 5
    Escalera
    IncrementarTotal
    Recalcular
  MostrarResultado
fin
```

A continuación se analiza la ejecución de este programa.

Una vez declaradas las variables, las constantes y los procesos, la ejecución comienza con la primera instrucción del programa principal después de comenzar. Esta instrucción es `Iniciar` (posiciona al robot en (1,1)), continúa con una llamada al proceso `Inicializar`, la cual causa que la ejecución secuencial del programa principal quede temporariamente suspendida y sea ejecutado el código del proceso `Inicializar`. Cuando el proceso `Inicializar` termina, el control retorna al punto siguiente desde donde fue llamado y la ejecución secuencial continúa. Luego se encuentra la estructura `repetir`, que permite que se ejecute 5 veces el bloque de instrucciones que aparece a continuación. Se ejecuta, entonces, la llamada al proceso `Escalera`, la que produce que la ejecución secuencial se suspenda nuevamente y comiencen a ejecutarse las instrucciones de este proceso. Una vez que este proceso termina, el flujo de control retorna a la instrucción siguiente de la que se lo llamó desde el programa principal, y retoma nuevamente la ejecución secuencial. Continua así hasta finalizar con la ejecución del último proceso que retorna el control al programa principal, para comenzar otra vez con este bloque de tres instrucciones (llamadas a procesos) hasta completar las 5 veces de la estructura `repetir`, luego se invoca `MostrarResultado` que procede de igual manera. Finalmente, ejecuta la instrucción siguiente `end` con lo cual termina la ejecución del programa.

Es importante destacar algunos aspectos de esta solución:

- Los procesos están declarados antes del comienzo del programa principal. Esto se debe a una regla general que dice que todos los procesos que utilice un programa, deben ser conocidos antes de ser usados.
- Si se compara la última solución con la anterior se puede observar que no hay código nuevo, excepto las llamadas a los procesos.
- El programa principal queda como un resumen de alto nivel del programa total. Como resultado final, el programa principal representa una especificación del programa. Este programa principal es simple y corto ya que la tarea de detalle queda oculta (*hidden*) en los módulos del programa.

2.5.2 Alcance de los datos

La importancia del ocultamiento de los datos (*Data Hidding*)

En el desarrollo de un sistema complejo con gran número de módulos, realizados por diferentes programadores, debe tenerse en cuenta que cada uno de ellos escribirá el código correspondiente a un módulo. Esto implica que cada cual define sus propios datos, con identificadores apropiados y como consecuencia de ello se podrían observar algunos inconvenientes tales como:

- Demasiados identificadores.

- Conflictos entre los nombres de los identificadores de cada programador.
- Integridad de los datos lo que implica que se puedan utilizar datos que tengan igual identificador pero que realicen funciones diferentes.
- *Side Effects* no previstos.

La solución para reducir estos problemas se logra con una combinación de ocultamiento de datos (*Data Hidding*) y uso de "Parámetros". Estos conceptos se discutirán a continuación.

2.5.2.1 Datos locales y datos globales

La diferencia entre dato global y dato local es simple: los datos globales son aquellos que se declaran en la sección de declaración del programa principal. En el ejemplo 2.21, todos los datos son globales.

Los datos locales son aquellos que se declaran en la sección de declaración de un módulo individual.

2.5.2.2 Ocultamiento y protección de datos ("*Data Hidding*")

El concepto de *Data Hidding* se utiliza aquí para significar que todo dato que es relevante para un módulo debe ocultarse de los otros módulos. De esta manera se evita que en el programa principal se declaren datos que sólo son relevantes para algún módulo en particular y, además, se protege la integridad de los datos.

Ejemplo 2.23

Si se vuelve al ejemplo planteado anteriormente, se observa que algunos datos pueden ser declarados en forma local y otros seguirán siendo globales.
En el proceso Escalera se puede observar que la variable Pasos puede ser declarada local al módulo, ya que solo es usada en dicho módulo.
a continuación se reescribe el ejemplo planteado, utilizando variables locales y globales.

```
{ El robot está en (1,1) orientado hacia el norte }

programa Recorrido

variables
  Escalones: numero      { Indica la cantidad de escalones de cada
                           recorrido }
  Papeles: numero        { Indica la cantidad de papeles de cada
                           recorrido }
  TotalPapeles: numero   { Indica la cantidad total de papeles de
                           todos los recorridos }
```

```
    Avenida: numero  { Indica la avenida en la que comienza el recorrido }

  procesos
    proceso Inicializar
    comenzar
      Escalones := 9
      TotalPapeles := 0
      Avenida := 1
    fin

    proceso Izquierda
    comenzar
      repetir 3
        derecha
    fin

    proceso Escalera
    variables
      Pasos: numero     { Indica los pasos dados en cada recorrido }
    comenzar
      Papeles := 0
      Pasos := 0
      repetir Escalones
        mientras HayPapelEnLaEsquina
          tomarPapel
          Papeles := Papeles + 1
        mover
        Pasos := Pasos + 1
        derecha
        mientras HayPapelEnLaEsquina
          tomarPapel
          Papeles := Papeles + 1
        mover
        Pasos := Pasos + 1
        Izquierda
        { juntar los papeles de la última esquina del recorrido }
        mientras HayPapelEnLaEsquina
          tomarPapel
          Papeles := Papeles + 1
        { Informar la cantidad de pasos dados }
        Informar(Pasos)
    fin

    proceso IncrementarTotal
    comenzar
      TotalPapeles := TotalPapeles + Papeles
    fin
```

BIBLIOTECA
FAC. DE INFORMATICA
U.N.L.P.


```
    proceso Recalcular
    comenzar
      Avenida := Avenida + 2
      Pos(Avenida, 1)
      Escalones := Escalones - 2
    fin

    proceso MostrarResultado
    comenzar
      Informar(TotalPapeles)
    fin

  comenzar { programa principal }
    iniciar
    Inicializar
    repetir 5
      Escalera
      IncrementarTotal
      Recalcular
    MostrarResultado
  fin
```

Si se observa el código escrito, se puede notar que no se ha realizado ningún cambio en los datos utilizados, solo se ha cambiado el lugar donde se declaran. Esto produce dos consecuencias lógicas: permite una mejor organización de los datos y logra que los mismos estén protegidos.

Cuando se hace referencia a una mejor organización de los datos, se quiere significar que solo se declaran datos globales en el caso que los mismos sean utilizados por más de un módulo. Para aquellos datos que solo son utilizados en un módulo particular se declaran locales a dicho módulo, de manera que permanecen ocultos (*hidden*) para los demás módulos alcanzando, de esta manera, la protección de tales datos y, en consecuencia, la integridad de los mismos.

2.5.2.3 Alcance de los datos

Hasta aquí se han utilizado los términos local y global de manera absoluta, pero se debe considerar que los módulos pueden anidarse. Así como se declaran módulos dentro del programa principal, pueden declararse módulos dentro de otros módulos. Por lo tanto, deben aplicarse dos reglas muy importantes que gobiernan el alcance de los identificadores, y que se mencionan a continuación:

- El alcance de un identificador es el bloque de programa donde se lo declara.
- Si un identificador declarado en un bloque es nuevamente declarado en un bloque interno al primero, el segundo bloque es excluido del alcance de la primera sección, (véase figura 2.10).

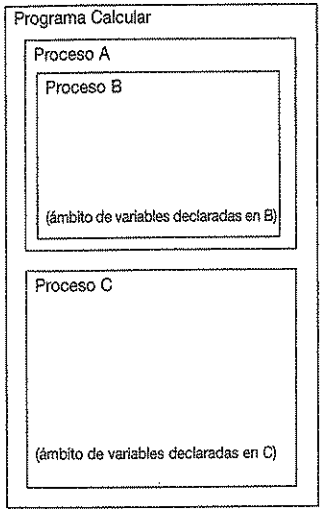


Figura 2.10

2.5.2.4 Parámetros

Los parámetros son variables que tienen como característica principal transferir información entre los módulos.

Cuando un programa está bien organizado e implementado utiliza parámetros para todos los datos compartidos entre módulos. Se puede adoptar, como regla general, que toda la información que viaja hacia o desde un módulo debe hacerse a través de parámetros, en lugar de utilizar variables globales a las que se accede desde todos los módulos.

Hay dos tipos de parámetros que interesan, los parámetros "pasados por valor" y los parámetros "pasados por referencia".

Cuando existen datos compartidos entre módulos, una solución es que un módulo pase una copia de esos datos al otro. En este caso, el parámetro que se pasa se denomina parámetro "pasado por valor". El módulo que recibe esta copia no puede efectuar ningún cambio sobre el dato original.

Los parámetros "pasados por referencia", a diferencia de los parámetros "pasados por valor", no envían una copia del valor del dato, sino que envían la dirección donde se encuentra el dato, con lo cual tanto el programa o el proceso que llama, como así también el proceso llamado, puede acceder a dicho dato.

A continuación se analizan algunas de las razones que permitirán explicar por qué es preferible el uso de parámetros al uso de variables globales, obviamente en aquellas situaciones en las que es posible elegir alguna de las dos alternativas:

Integridad de los datos

En este caso es necesario conocer qué datos usa cada módulo. Este problema puede solucionarse, en parte, declarando tales datos como locales al módulo, es decir, ocultándolos de los demás módulos, (esto se presenta en la solución planteada anteriormente). Sin embargo, se puede observar que el problema no queda resuelto cuando los datos deben ser compartidos por distintos módulos. La solución, entonces, implica conocer qué datos globales requiere cada uno de los módulos y qué datos globales son compartidos por los distintos módulos. Además, es importante saber si un módulo particular utiliza el dato solo de "lectura" o bien lo modifica. Es aquí, donde se utilizan los parámetros.

Uso de parámetros para proteger los datos

Si una variable es local a un módulo se puede asegurar que cualquier otro módulo fuera del alcance de esa variable no la puede "ver" y, por lo tanto, no la puede modificar. Dado que las variables globales pueden ser accedidas desde distintas partes del programa, no se puede garantizar que accidentalmente no se modifiquen. La solución al problema será utilizar parámetros "pasados por valor".

Volviendo al proceso Escalera, es necesario reconocer la cantidad de escalones para cada uno de los recorridos. Se puede, entonces, reescribir la solución de la siguiente manera:

Ejemplo 2.24

```
{ El robot está en (1,1) orientado hacia el norte }

programa Recorrido

variables
  Escalones: numero      { Indica la cantidad de escalones de cada
                           recorrido}
  Papeles: numero        { Indica la cantidad de papeles de cada
                           recorrido}
  TotalPapeles: numero   { Indica la cantidad total de papeles de
                           todos los recorridos}
  Avenida: numero        { Indica la avenida en la que comienza el
                           recorrido}

procesos
  proceso Inicializar
  comenzar
    Escalones := 9
    TotalPapeles := 0
    Avenida := 1
  fin
```

```
proceso Izquierda
comenzar
  repetir 3
    derecha
fin

proceso Escalera (E Esc: numero)
variables
  Pasos: numero              {Indica los pasos dados en cada
                              recorrido }
comenzar
  Papeles := 0
  Pasos := 0
  repetir Esc
    mientras HayPapelEnLaEsquina
      tomarPapel
      Papeles := Papeles + 1
    mover
    Pasos := Pasos + 1
    derecha
    mientras HayPapelEnLaEsquina
      tomarPapel
      Papeles := Papeles + 1
    mover
    Pasos := Pasos + 1
    Izquierda                {juntar los papeles de la última
                              esquina del recorrido}
  mientras HayPapelEnLaEsquina
    tomarPapel
    Papeles := Papeles + 1  {Informar la cantidad de pasos dados }
  Informar(Pasos)
fin

proceso IncrementarTotal(E Pap: numero)
comenzar
  TotalPapeles := TotalPapeles + Pap
fin

proceso Recalcular
comenzar
  Avenida := Avenida + 2
  Pos(Avenida, 1)
  Escalones := Escalones - 2
fin

proceso MostrarResultado(E TotPap: numero)
comenzar
  Informar(TotPap)
fin
```



```
{programa principal}
comenzar
  iniciar
  Inicializar
  repetir 5
    Escalera(Escalones)
    IncrementarTotal(Papeles)
    Recalcular
  MostrarResultado(TotalPapeles)
fin
```

Se puede observar que en esta solución se ha cambiado el encabezamiento (*Header*) del proceso Escalera y la llamada a dicho proceso. Al cambiar el encabezamiento, este proceso espera recibir como dato de entrada la variable Escalón. Este dato le permitirá ejecutar el recorrido esperado. Por otro lado, en la llamada al proceso se especifica el dato que debe enviarse al proceso como parámetro. El número de parámetros enviados (en la invocación) y de parámetros recibidos (en el *header*) deben coincidir. Los parámetros enviados se denominan parámetros "actuales" y los parámetros que aparecen en el encabezamiento del proceso se definen como parámetros "formales". La asociación entre los parámetros actuales y los parámetros formales se hace por su posición en la lista respectiva de parámetros, así el primer parámetro enviado en la lista de parámetros actuales se asocia al primer parámetro de la lista de parámetros formales, y así sucesivamente.

Por ejemplo, en el lenguaje Pascal, debido a su característica de "fuertemente tipado", además de coincidir en el número de parámetros actuales y formales, deben coincidir en el tipo de dato asociado a cada uno de los parámetros actuales y formales respectivamente ya que, en caso contrario, se producen errores en tiempo de compilación. Estos conceptos se analizarán con más detalles en el capítulo 4.

Uso de parámetros para retornar datos

El parámetro "pasado por valor" es útil para proteger los datos. El enviar una copia de la información evita que el módulo lo modifique. Sin embargo, existen situaciones en las que se quiere conocer qué modificaciones se hicieron sobre los datos. Para retornar datos al módulo que hizo el llamado se utilizan los parámetros "pasados por referencia". Cuando se utilizan parámetros en un módulo se deben diferenciar cuales son "por valor" y cuales "por referencia" ya que los datos se pasan de manera diferente. Para distinguir los parámetros "pasados por referencia", por ejemplo, en Pascal, se le antepone la palabra `Var`.

Ejemplo 2.25

Por ejemplo, el proceso Escalera debe devolver la cantidad de papeles encontrados en el recorrido, el proceso Inicializar debe retornar los valores iniciales para comenzar el recorrido. Se puede, entonces, reescribir la solución de la siguiente manera:

```
{ El robot está en (1,1) orientado hacia el norte }
programa Recorrido
variables
  Escalones: numero      { Indica la cantidad de escalones de cada
                           recorrido}
  Papeles: numero        { Indica la cantidad de papeles de cada
                           recorrido}
  TotalPapeles: numero   { Indica la cantidad total de papeles de
                           todos los recorridos}
  Avenida: numero        { Indica la avenida en la que comienza el
                           recorrido}
procesos
  proceso Inicializar(S Esc: numero; S TotPap: numero; S Ave: numero)
  comenzar
    Esc := 9
    TotPap := 0
    Ave := 1
  fin

  proceso Izquierda
  comenzar
    repetir 3
      derecha
  fin

  proceso Escalera (E Esc: numero; S Pap: numero)
  variables
    Pasos: numero        { Indica los pasos dados en cada recorrido }
    PapAux: numero
  comenzar
    PapAux := 0
    Pasos := 0
    repetir Esc
      mientras HayPapelEnLaEsquina
        tomarPapel
        PapAux := PapAux + 1
      mover
      Pasos := Pasos + 1
      derecha
      mientras HayPapelEnLaEsquina
        tomarPapel
        PapAux := PapAux + 1
      mover
      Pasos := Pasos + 1
      Izquierda          { juntar los papeles de la última esquina del
                           recorrido}
    mientras HayPapelEnLaEsquina
      tomarPapel
```



```
      PapAux := PapAux + 1
    Pap := PapAux  { Informar la cantidad de pasos dados }
    Informar(Pasos)
  fin

  proceso IncrementarTotal(E Pap: numero)
  comenzar
    TotalPapeles := TotalPapeles + Pap
  fin

  proceso Recalcular
  comenzar
    Avenida := Avenida + 2
    Pos(Avenida, 1)
    Escalones := Escalones - 2
  fin

  proceso MostrarResultado(E TotPap: numero)
  comenzar
    Informar(TotPap)
  fin
{programa principal}
comenzar
  iniciar
  Inicializar(Escalones, TotalPapeles, Avenida)
  repetir 5
    Escalera(Escalones, Papeles)
    IncrementarTotal(Papeles)
    Recalcular
  MostrarResultado(TotalPapeles)
fin
```

Hasta aquí la solución presenta todos los datos de entrada como parámetros pasados por valor y los datos de salida como parámetros por referencia.

Uso de parámetros para enviar datos en ambos sentidos

Como se vio, los parámetros "pasados por valor" se utilizan como parámetros de entrada y los "pasados por referencia" como parámetros de salida. Existe una tercera posibilidad que es cuando un parámetro es usado como entrada y salida. Esto ocurre cuando, por ejemplo, un dato es enviado como entrada a un módulo, luego el dato es modificado en ese módulo y, por último, el dato modificado debe retornar al módulo que hizo el llamado. En estas circunstancias el parámetro debe ser declarado como parámetro "pasado por referencia", ya que de otro modo queda protegido y no se conoce la modificación fuera del módulo. Como conclusión, se utilizan parámetros "pasados por referencia" en dos circunstancias: cuando se necesita que el parámetro sea de "salida" o bien, cuando se necesita que el parámetro sea de "entrada/salida". No hay ninguna diferencia en cuanto al mecanismo empleado para estos dos casos, sin em-

bargo, es importante destacar, que la distinción debe hacerse explícita en la documentación del programa correspondiente.

Ejemplo 2.26

En el ejemplo se puede observar que en el proceso `IncrementarTotal`, la variable `Total Papeles` es de entrada/salida, y en el proceso `Recalcular`, las variables Avenida y Escalones también son de entrada/salida.
Se puede, entonces, reescribir la solución considerando ahora en forma completa todos los datos de entrada, de salida y los datos de entrada/salida de la siguiente manera:

```
programa Recorrido
procesos
  proceso Inicializar(S Esc: numero; S TotPap: numero; S Ave: numero)
  comenzar
    Esc := 9
    TotPap := 0
    Ave := 1
  fin

  proceso Izquierda
  comenzar
    repetir 3
      derecha
  fin

  proceso Escalera (E Esc: numero; S Pap: numero)
  variables
    Pasos: numero { Indica los pasos dados en cada recorrido }
    PapAux: numero
  comenzar
    PapAux := 0
    Pasos := 0
    repetir Esc
      mientras HayPapelEnLaEsquina
        tomarPapel
        PapAux := PapAux + 1
      mover
      Pasos := Pasos + 1
      derecha
      mientras HayPapelEnLaEsquina
        tomarPapel
        PapAux := PapAux + 1
      mover
      Pasos := Pasos + 1
      Izquierda   { juntar los papeles de la última esquina del
                    recorrido}
    mientras HayPapelEnLaEsquina
```



```
        tomarPapel
        PapAux := PapAux + 1
    Pap := PapAux   { Informar la cantidad de pasos dados }
    Informar(Pasos)
  fin

  proceso IncrementarTotal(E Pap: numero; ES TotPap: numero)
  comenzar
    TotPap := TotPap + Pap
  fin

  proceso Recalcular(ES Ave: numero; ES Esc: numero)
  comenzar
    Ave := Ave + 2
    Pos(Ave, 1)
    Esc := Esc - 2
  fin

  proceso MostrarResultado(E TotPap: numero)
  comenzar
    Informar(TotPap)
  fin

variables
  Escalones: numero     { Indica la cantidad de escalones de cada
                          recorrido}
  Papeles: numero       { Indica la cantidad de papeles de cada
                          recorrido}
  TotalPapeles: numero  { Indica la cantidad total de papeles de
                          todos los recorridos}
  Avenida: numero       { Indica la avenida en la que comienza el
                          recorrido}

{programa principal}
comenzar
  iniciar
  Inicializar(Escalones, TotalPapeles, Avenida)
  repetir 5
    Escalera(Escalones, Papeles)
    IncrementarTotal(Papeles, TotalPapeles)
    Recalcular(Avenida, Escalones)
  MostrarResultado(TotalPapeles)
fin
```

Observación: el lector debe notar que la estructura esquemática presentada aquí varía de la presentada en Apéndice A. Esto se debe a detalles de implementación del robot.

3

Tipos de datos simples

Objetivos

La representación de la información es vital para trabajar con computadoras. En los capítulos anteriores se discutió la forma de analizar y definir un algoritmo, el problema radica ahora en dotar a los algoritmos de elementos que permitan manipular información. Esta información puede ser variada, desde datos numéricos (edades, salarios, cantidad de materias aprobadas) o nombres de personas, hasta todos los datos que componen un sistema bancario.

En este capítulo se definen los **tipos de datos simples**, existentes en todos los lenguajes de programación, que permiten manipular las representaciones más sencillas de información. Los objetivos del capítulo están centrados en comprender estas representaciones elementales, analizando el conjunto de operaciones válidas definidas para cada una de ellas. La representación interna, dentro de la memoria de la computadora, de los distintos tipos de datos simples no forma parte del objetivo de este libro, y para los interesados en analizarla se sugiere ver (Brookshear, 1995), (Anazagasti, 1993) y (Georgia Tech, 1993). Los detalles del lenguaje Pascal utilizado en la ejemplificación se pueden ver en (Leestma, 1984).

Utilidad del uso de parámetros

Al escribir programas el uso de parámetros independiza a cada programador del nombre de los identificadores que usen los demás. Por otro lado, al probar un código es muy útil conocer la interfaz entre los módulos y generar pruebas asignando valores a cada uno de los parámetros que intercambian. Es sabido, también, que los programas requieren mantenimiento. Si estos programas se han diseñado en forma modular, la tarea de mantenimiento se realizará a menor costo. Esto es debido, en parte, a que la tarea de localizar errores será más fácil y se podrá determinar rápidamente cuáles son los módulos que se verán afectados por la corrección de esos errores.

Reuso de código

El uso de parámetros permite separar el nombre del dato, del dato en sí mismo, lo que permite que el mismo código sea utilizado en diferentes partes de un programa, simplemente cambiando la lista de parámetros actuales.

Dado que, a partir de aquí, los ejemplos de aplicación se escribirán en lenguaje Pascal, es importante aclarar que este lenguaje provee dos maneras de implementar módulos: ***Procedure*** (procedimiento) y ***Function*** (función).

Los procedimientos permiten devolver más de un dato al programa o módulo que lo invoca, y el flujo de control del programa retorna a la instrucción siguiente a la del llamado.

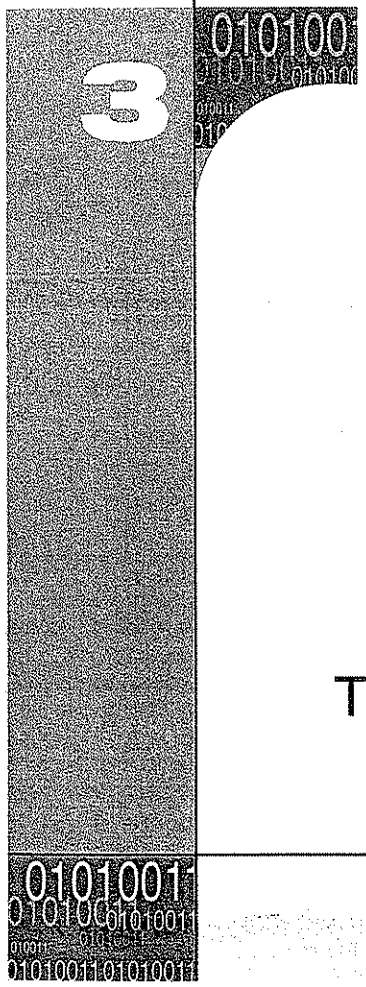
Las funciones permiten devolver un único valor al programa o módulo de programa que la invoca y el flujo de control del programa retorna a la misma instrucción que efectuó el llamado.

Más adelante, se estudiarán y presentarán cada uno de estos módulos con mayor detalle.

Conclusiones

Se pueden puntualizar algunas reglas básicas de trabajo:

1. Se debe adoptar un método de descomposición del problema *top down*, o sea: analizar el problema, dividirlo en subproblemas y reiterar el método con cada subprograma (módulo) hasta reducir la tarea a una función o procedimiento bien definido.
2. Se deben usar datos locales siempre que sea posible, y utilizar datos globales cuando no sea posible utilizar datos locales.
3. La comunicación entre módulos debe definirse a través de parámetros. Los módulos no deberían acceder a los datos globales, sino recibir los datos necesarios como parámetros.
4. Es imprescindible documentar cada uno de los módulos, haciendo algún comentario acerca del objetivo del módulo y precondiciones y poscondiciones para los datos utilizados.
5. Se debe notar que las poscondiciones indican el efecto sobre los datos del proceso. Se debe indicar cuál es el efecto del módulo.



Capítulo 3

Tipos de datos simples

3.1 Tipos de datos

Los algoritmos generalmente operan sobre datos de distinta naturaleza, tales como números, letras, símbolos, etc. Por lo tanto, los programas que implementan dichos algoritmos, necesitan alguna manera de representarlos.

Definición

Un tipo de dato es una clase de objetos ligados a un conjunto de operaciones para crearlos y manipularlos.

Los tipos de datos se caracterizan por:

- un rango de valores posibles,
- un conjunto de operaciones realizables sobre ese tipo,
- su representación interna.

Al definir un tipo de dato lo que se está indicando es la clase de valores que pueden tomar sus elementos y las operaciones que se pueden realizar sobre ellos.

Cada tipo de datos se representa de distinta forma en la computadora. A nivel de máquina, un dato es solo una secuencia de ceros y unos (secuencia de bits). Los lenguajes de alto nivel permiten basarse en abstracciones e ignorar los detalles de la representación interna.

Los tipos de datos simples son:

- Numérico
- Lógico
- Carácter

Algunos lenguajes, como por ejemplo Pascal, permiten definir nuevos tipos, denominados tipos definidos por el usuario; hay una sección, dentro de este capítulo, dedicada a dicho tema.

3.1.1 Tipo de dato numérico

El **tipo de dato numérico** es el conjunto de los valores numéricos que pueden representarse de dos formas:

- Enteros
- Reales

3.1.1.1 Enteros

El tipo de dato numérico más simple de todos es el entero. Los elementos del tipo son:

..., -3, -2, -1, 0, 1, 2, 3, ...

Dado que una computadora tiene memoria finita, la cantidad de valores enteros que se pueden representar sobre ella son finitos, por esto se deduce que existe un número entero máximo y otro mínimo.

La cantidad de valores que se pueden representar depende de la cantidad de memoria (bits) que se utilicen para representar un entero.

Ejemplo 3.1

Hay sistemas de representación numérica que utilizan 16 dígitos binarios (bits) para almacenar en memoria cada número entero, permitiendo un rango de valores enteros entre -215 y +215. Otros sistemas utilizan 32 bits, por lo que el rango es entre -231 y +231.

En general, para cada computadora existe el entero `maxint`, tal que todo número entero n puede ser representado si,

```
- maxint <= n <= maxint
```

donde `maxint` identifica al número entero más grande que se puede representar con la cantidad de dígitos binarios disponibles.

3.1.1.2 Reales

El tipo de dato real es una clase de dato numérico que permite representar números decimales. Los valores fraccionarios forman una serie ordenada, desde un valor mínimo negativo, hasta un valor máximo determinado por la norma IEEE 754, de 1985, pero los valores no están distribuidos de manera uniforme en ese intervalo, como sucede con los enteros.

Se debe tener en cuenta que el tipo de dato real tiene una representación finita de los números reales; dicha representación tiene una precisión fija, es decir, un número fijo de dígitos significativos. Esta condición es la que establece una diferencia con la representación matemática de los números reales. En este caso se tienen infinitos números diferentes, en tanto que la cantidad de representaciones del tipo de dato real está limitada por el espacio en memoria disponible.

La representación para números reales se denomina **coma flotante**. Esta es una generalización de la conocida notación científica, utilizada por cualquier tipo de calculadora, y consiste en definir cada número como una mantisa (parte decimal) y un exponente (posición de la coma).

Ejemplo 3.2

Sea el número 89412950000000000. Su representación científica en cualquier calculadora es $8{,}941295 \times 10^{16}$
La mantisa del número real representado en la computadora en este caso es: 0,8941295 y el exponente 17. Nótese que, a diferencia de la representación anterior, la coma se ubica inmediatamente a la izquierda del primer dígito significativo y, por lo tanto, la magnitud del exponente se incrementa en 1.
Si el número es 0,0000000000356798, su representación utilizando notación científica es: $3{,}56798 \times 10^{-11}$
La mantisa es 0,356798 y el exponente es -10.
Se observa que un exponente positivo lleva a desplazar la coma decimal tantos lugares hacia la derecha como lo indica su magnitud, mientras un exponente negativo implica un desplazamiento hacia la izquierda.

3.1.1.3 Operaciones sobre datos numéricos

Se ha presentado hasta el momento la forma de representación y los valores posibles de cada tipo de dato numérico. Como se indica en la definición de tipo de dato, los mismos cuentan con un conjunto de operaciones posibles.

Las operaciones válidas para el tipo de dato numérico son: **suma** (+), **resta** (-), **multiplicación** (*), **división** (/), **división entera** (div) y **módulo** (mod). La operación `div`, de Pascal, da como resultado el cociente de una división, mientras que `mod`, también de Pascal, da el resto de una división.

Los símbolos +, -, *, /, `div` y `mod` son conocidos como **operadores aritméticos**. En la expresión 5 + 4, los valores 5 y 4 se denominan **operandos**, y al valor de la expresión se lo conoce como **resultado**.

Ejemplo 3.3

3 + 2 = 5	3,14 + 2,34 = 5,48
3 - 1 = 2	4,15 - 2,13 = 2,02
7 - 9 = -2	5,1 - 8,2 = - 3,1
4 * 5 = 20	5,12 * 3,4 = 17,408
8 * (-2) = -16	3,1 / 2 = 1,55
5 div 2 = 2	7 mod 5 = 2

No todas las operaciones definidas son aplicables a todos los tipos de datos numéricos. La suma, resta y multiplicación aceptan tanto operandos enteros como reales y, por lo tanto, su resultado podrá ser entero o real. La operación de división solamente retorna resultados de tipo real. Por último, `mod` y `div` solo valen con operandos enteros y su resultado es entero.

Cuando se están realizando operaciones, utilizando tanto tipos de datos enteros como reales, puede ocurrir que los resultados de alguna de ellas supere el máximo valor binario permitido por la representación, en dicho caso se está ante la presencia de un caso extremos que se denomina ***overflow***. Este caso extremo es tratado en forma diferente. Mientras que algunos sistemas lo detectan como error, (presentado un mensaje con esa situación), otros lo ignoran, convirtiendo el valor resultado en un valor válido dentro del rango pero que, obviamente, no será el resultado esperado de la operación planteada.

Puede suceder, además, que cuando se realizan operaciones se intente representar, como resultado de las mismas, valores demasiado pequeños. En este caso extremo el resultado está por debajo del límite permitido y se está ante la presencia de un ***underflow***. Dichos *underflows* son tratados en forma similar a lo definido anteriormente; esto es, algunos sistemas los interpretan como error, en tanto que otros le dan una solución de compromiso.

Ejemplo 3.4

Si se está utilizando el lenguaje Pascal y se solicita la siguiente operación:
`write( ' 32767 + 1 = ', 32767+1 )`
el resultado obtenido indicará -32767, a diferencia del valor que se esperaba 32768. Esto se debe a la cantidad de bits destinados por Pascal para la representación de los tipos de datos enteros, que son 16. El máximo número representable es 32767, por lo tanto, al sumarle uno se produce un caso extremo: *overflow*. Pascal lleva sus casos extremos a una representación válida dentro de su rango y, por este motivo, el resultado es el indicado.

Las expresiones que tienen dos o más operandos requieren reglas matemáticas que permitan determinar el orden de las operaciones. El **orden de precedencia** para la resolución, ya conocido, es:

operadores *, /
operadores +, -
operadores `div` y `mod`.

En caso que el orden de precedencia natural deba ser alterado, es posible la utilización de paréntesis dentro de la expresión.

Ejemplo 3.5

6 + 8 * 5 = 6 + 40 = 46
(6 + 8) * 5 = 14 * 5 = 70
5 * 2 + 7 + 4 * 3 = 10 + 7 + 12 = 17 + 12 = 29

Además de los operadores matemáticos mencionados, el tipo de dato numérico posee operadores relacionales que permiten comparar valores. Dichas relaciones son la igualdad (=), desigualdad (<>) y de orden (<, <=, >, >=). El resultado es del tipo de dato lógico, que se describe a continuación.

3.1.2 Tipo de dato lógico

El **tipo de dato lógico**, también llamado **booleano,** en honor al matemático británico George Boole (quien desarrolló el Álgebra Lógica o de Boole), es un dato que puede tomar un valor entre un conjunto formado por dos posibles. Dichos valores son:

- verdadero (`true`)
- falso (`false`)

Se utiliza en casos donde se representan dos alternativas a una condición. Por ejemplo, si se debe determinar si un valor es primo; la respuesta será verdadera *(true)* si el número es divisible solamente por sí mismo y la unidad; en caso contrario, si tiene algún otro divisor, la respuesta será falsa (`false`).

Operaciones sobre datos lógicos

Los operadores lógicos o booleanos básicos son:

- negación (`not`),
- conjunción (`and`),
- disyunción (`or`).

El resultado de estas operaciones es el correspondiente a las conocidas tablas de verdad. Dados p y q (datos lógicos), la tabla de verdad de la negación se presenta como Tabla 3.1, la correspondiente a la conjunción en Tabla 3.2; en tanto que la Tabla 3.3. presenta la operación de disyunción.

p	not p
verdadero	falso
falso	verdadero

Tabla 3.1

p	q	p and q
verdadero	verdadero	verdadero
verdadero	falso	falso
falso	verdadero	falso
falso	falso	falso

Tabla 3.2

p	q	p or q
verdadero	verdadero	verdadero
verdadero	falso	verdadero
falso	verdadero	verdadero
falso	falso	falso

Tabla 3.3

Los operadores relacionales mencionados anteriormente, que permiten comparar dos valores, dan como resultado un valor lógico.

Ejemplo 3.6

(8 < 4)	da como resultado falso
(2.5 * 4 = 10)	da como resultado verdadero
(8 > 3)	da como resultado verdadero
`not` (8 > 3)	da como resultado falso
(7 > 2.4), (9 = 3)	son expresiones verdaderas y falsa respectivamente
(7 > 2.4) and (9 = 3)	una expresión que da un resultado falso
(7 > 2.4) or (9 = 3)	es una expresión que da un resultado verdadero

Existe un orden de precedencia para los operadores lógicos, del mismo modo que para los matemáticos. Dicho orden es:

1. operador `not`
2. operador `and`
3. operador `or`

3.1.3 Tipo de dato carácter

Un **tipo de dato carácter** proporciona objetos de la clase de datos que contiene solo un elemento como su valor. Este conjunto de elementos está establecido y normatizado por un estándar llamado ASCII *(American Standard Code for Information Interchange)*, el cual establece cuales son los elementos y el orden de precedencia entre los mismos. Los elementos son las letras, número y símbolos especiales disponibles en el teclado de la computadora y algunos otros elementos gráficos. Cabe acotar que el código ASCII no fue único, pero es el más utilizado internacionalmente.

Ejemplos de los elementos carácter son:

- Letras minúsculas: 'a', 'b', 'c', ..., 'y', 'z'.
- Letras mayúsculas: 'A', 'B', 'C', ..., 'Y', 'Z'.
- Dígitos: '0', '1', '2', ..., '8', '9'.
- Caracteres especiales: '!', '@', '#', '$', '%', °.

Se debe tener en cuenta que no es lo mismo el valor entero 0 que el símbolo carácter '0'.

Un valor del tipo de dato carácter es **solo uno** de los símbolos mencionados. Más adelante se describe el tipo de dato cadena de caracteres *(string)*, generalización del tipo de dato carácter.

3.1.3.1 Operaciones sobre datos carácter

Los operadores relacionales descriptos en el tipo de dato numérico, pueden utilizarse también sobre los valores del tipo de dato carácter. Esto es, dos valores de tipo carácter se pueden comparar por =, <>, >, <, >=, <=; el resultado de cualquiera de ellos es un valor de tipo de dato lógico.

La comparación de dos caracteres es posible dado que el código ASCII define para cada símbolo un valor en su escala. De esta manera, al comparar dos símbolos, para determinar el resultado se utiliza el valor dado por el código.

Dentro del código ASCII los valores de los dígitos son menores que los valores de las letras mayúsculas, y estos a su vez menores que los de las letras minúsculas. Los valores dentro del código para los símbolos especiales, o bien son menores que los dígitos, o bien mayores que las letras minúsculas.

Ejemplo 3.7

('a' = 'A')	da como resultado falso
('c' < 'Z')	da como resultado falso
('c' < 'z')	da como resultado verdadero
('X' > '5')	da como resultado verdadero
(' ' < 'H')	da como resultado verdadero
	Observación: ' ' representa un espacio en blanco
('4' = 4)	no puede evaluarse pues los operandos son de tipos distintos

La Figura 3.1 resume lo expuesto sobre tipos de datos hasta el momento.

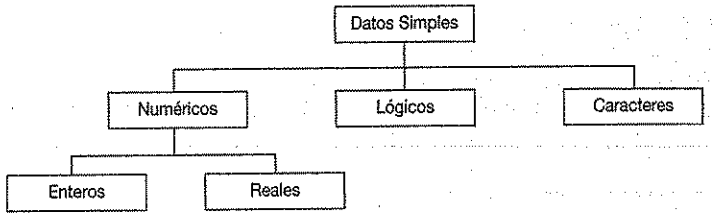


Figura 3.1

3.2 Constantes y variables

Como se definió en el capítulo 2, durante la ejecución de un programa es necesario manipular información, además, la misma puede variar continuamente. Por este motivo, es imprescindible contar con un elemento que permita variar la información que se maneja en cada momento, el resultado es la definición del concepto de **variables**.

Además de variables, en un programa puede manejarse información que no se altera durante toda la ejecución del mismo, es decir, que permanece **constante**.

Definición

Una constante en un programa denota un dato que no cambia durante la ejecución. Este tipo de dato puede ser numérico, lógico o carácter.

Tanto las constantes como las variables pueden contener información de alguno de los tipos de dato mencionados anteriormente.

3.2.1 Declaraciones

Se denominan identificadores a los nombres descriptivos que se asocia a los objetos (constantes y variables) para abstraer dentro del programa su dirección real en memoria y su valor. Los identificadores están formados por letras, dígitos y algunos símbolos especiales, con la restricción que el primer carácter debe ser una letra.

La mayoría de los lenguajes de programación poseen dentro de su sintaxis instrucciones que permiten definir las constantes y las variables que se utilizan dentro de un programa asociándolo con un identificador.

En Pascal, las constantes deben ser declaradas antes de ser utilizadas. Siguiendo su notación, la declaración de constantes se realiza mediante la palabra clave `const`, de la forma:

```
const nombre_de_constante = valor;
```

donde `nombre_de_constante` es el identificador que representa el nombre de la constante. El tipo de dato de la constante queda definido implícitamente por el tipo de dato de valor.

Ejemplo 3.8

```
const n = 25 {n se asume un valor de tipo de dato numérico entero}
      pi = 3.1416 {pi se asume de tipo de dato numérico real}
      c = 'C'     {c se asume de tipo de dato carácter}
```

Recordar que el texto encerrado entre { } indica que es comentario para Pascal.

Por su parte, las variables se declaran utilizando la palabra clave **var**, de la forma:

```
var nombre_de_variable : tipo_de_variable
```

O, si hay varias variables del mismo tipo:

```
var nombre_de_variable_1, ..., nombre_de_variable_n : tipo_de_variable
```

Los diferentes `tipo_de_variable` posibles se corresponden con los tipos de datos vistos hasta el momento. Se detalla a continuación el nombre del tipo que se debe utilizar para cada caso:

- `integer` → tipo de dato numérico entero
- `real` → tipo de dato numérico real
- `boolean` → tipo de dato lógico
- `char` → tipo de dato carácter

Ejemplo 3.9

```
var cantidad:  integer; { cantidad podrá contener un número entero }
    total_cobrado: real; { total_cobrado podrá contener un número real }
    estado: boolean;     { estado podrá contener un valor lógico }
    letra: char;         { letra podrá contener un valor carácter }
```

Algunos lenguajes, como por ejemplo Pascal, exigen que se especifique a qué tipo pertenece cada una de las variables, y verifican que el tipo de los datos asignados a esa variable se correspondan con su definición. Esta clase de lenguajes se denomina **fuertemente tipados** *(strongly typed)*.

Otra clase de lenguajes, que verifica el tipo de las variables según su nombre, se denomina **auto tipados** *(self typed)*.

Existe una tercera clase de lenguajes que permiten que una variable tome valores de distinto tipo durante la ejecución de un programa. Esta se denomina **dinámicamente tipados** *(dinamically typed)*.

3.2.2 Asignaciones

Una de las sentencias básicas que poseen los lenguajes de programación es la asignación. Es una sentencia de la forma:

```
nombre_de_variable := expresión
```

Hay que recordar que la semántica consiste en evaluar la expresión que se encuentra a la derecha y almacenar el resultado en la variable que está a la izquierda.

En Pascal, para que una asignación sea válida, los tipos de dato del resultado de la expresión y de la variable a la cual se asigna deben ser iguales o compatibles. Se presenta más específicamente en el siguiente ejemplo.

Ejemplo 3.10

Sean las siguientes declaraciones de variables:

```
var a, b : integer;
c, d : real;
e, f : char;
g : boolean;
```

Las siguientes son asignaciones válidas:

```
a := 23;
b := -3;
c := 3.4;
e := 'f';      e toma el valor del carácter f (que se indica entre comillas).
f := e;        f toma el valor de la variable e, es decir el carácter f.
e := '2';      e, en este caso, toma el valor del dígito 2.
g := true;
g := (3 > 6);  g, en este caso, recibe el valor false pues 3 es menor que 6
```

Las siguientes no son asignaciones posibles:

```
a := 34.1;     a es variable entera y se le intenta asignar un valor real.
b := 'c';      b es variable entera y no puede recibir como valor una letra.
c := false;    c es variable real y no puede recibir un valor lógico
f := 3;        f es variable carácter y no puede recibir el valor 3, puede recibir el dígito '3'.
g := 6;        g es variable lógica y no puede recibir un valor entero.
```

3.3 Funciones predefinidas

Los lenguajes de programación, en su gran mayoría, poseen funciones que se pueden aplicar sobre los tipos de datos que se definieron anteriormente. En esta sección se describen y ejemplifican algunas de las ellas.

3.3.1 Funciones sobre los tipos de datos numéricos

En la tabla 3.4 se enumeran algunas de las funciones predefinidas en Pascal sobre los tipos de datos numéricos. En la primera columna se detalla la sintaxis de cada función, en la segunda columna se expresa la semántica de la misma. Las dos últimas columnas determinan el tipo de dato que cada función espera recibir y el que da como respuesta.

Función (sintaxis)	Descripción	Tipo de dato argumento	Tipo de dato resultado
Abs(x)	Valor absoluto de x	int o real	int o real
Exp(x)	Exponencial de x (ex)	int o real	real
ln(x)	Logaritmo natural de x (loge x)	int o real	real
Round(x)	Redondeo de x	real	int
Sqr(x)	Cuadrado de x	int o real	int o real
Sqrt(x)	Raíz cuadrada de x	int o real	real
Trunc(x)	Truncamiento de x	real	integer
Sin(x)	Seno del ángulo	Int o real	Int o real
Cos(x)	Coseno del ángulo	Int o real	Int o real
Tan(x)	Tangente del ángulo	Int o real	Int o real
Arcsin(x)	Arco seno del ángulo	Int o real	Int o real
Arccos(x)	Arco coseno del ángulo	Int o real	Int o real
Arctan(x)	Arco tangente del ángulo	Int o real	Int o real

Tabla 3.4

Ejemplo 3.11

```
abs(-7.4) = 7.4
round(1.4) = 1
round(2.9) = 3
round(-5.7) = -6
sqrt(81) = 9
sqr(9) = 81
trunc(4.9) = 4
trunc(4.2) = 4
```


Ejemplo 3.12

Sea el siguiente problema. Leer un conjunto de números hasta encontrar uno donde la suma de sus dígitos sea divisible por tres. Utilizar una función que reciba un número y determine si cumplen la propiedad enunciada.

Precondición:
- Lee números hasta encontrar uno donde la suma de sus dígitos sea múltiplo de 3.

Poscondición:
- Se imprime el número con la condición deseada.

El programa 3.1 presenta la solución del problema teniendo en cuenta las pre y las poscondiciones presentadas.

```
Program multiplo_tres;
  Function divisible_tres ( valor: integer): boolean;
    {dado un número, suma cada dígito para ver si es divisible por 3,
     retornando true o false segun corresponda }
    var suma,
        resto: integer;
    begin
      suma := 0;
      repeat
{1}     resto := valor mod 10; {toma el último dígito del número }
{2}     suma := suma + resto;  {acumula el valor del dígito }
        valor := valor div 10;  {toma el número sin el último dígito }
      until valor = 0;
{3}   divisible_tres := ( suma mod 3 = 0 )
    end;

  var numero: integer;
  begin
    {toma un número desde hasta encontar el múltiplo de tres }
    repeat
      read( numero );
    until divisible_tres( numero );
    writeln ('El numero encontrado es: ', numero );
  end.
```

Nótese en las instrucciones {1} y {2} pueden cambiarse por

```
suma := suma + valor mod 10;
```

De esta forma se evita utilizar la variable resto. En la instrucción {3} se asocia al nombre de la función el resultado que retornará la misma. Este se obtiene de comparar el resto de la división con 0; lo cual, como se sabe, da como resultado un valor de tipo de dato lógico. En el siguiente capítulo se presenta un análisis más detallado de la definición de funciones.

3.3.2 Funciones sobre el tipo de datos carácter

La tabla 3.5 presenta algunas de las funciones predefinidas en Pascal sobre el tipo de dato carácter y su semántica asociada.

función (sintaxis)	descripción	tipo de dato argumento	tipo de dato resultado
`Chr(x)`	retorna el caracter Ascii que se corresponde con el valor de x	`integer`	`char`
`Ord(x)`	retorna el valor Ascii del carácter que se indica en x	`char`	`integer`

Tabla 3.5

Ejemplo 3.13

```
ord('H') = 72
ord('2') = 50
chr(45) = '-'
chr(110) = 'n'
```

Ejemplo 3.14

Dado el problema de convertir una letra minúscula en una letra mayúscula.
El primer paso consisten determinar el código ascii de la letra involucrada. Para esto, se puede utilizar la función `ord`, que retorna dado un carácter dicho código. Una particularidad del código ascii es que la distancia numérica del código de minúsculas y mayúsculas es constante y puede obtenerse por diferencia entre el código numérico minúscula y mayúscula entre dos letras cualquiera.

```
distancia_minuscula_mayuscula := ord('a') - ord('A')
```

Si letra es la variable de tipo carácter, minúscula, que se desea convertir a mayúscula, una expresión del tipo

```
valor_codigo_ASCII := ord(letra) - ord('a') - ord('A')
```

permite convertir el código ascii de letra del valor correspondiente a la letra minúscula al valor correspondiente al valor de la letra en mayúscula.
Finalmente, falta convertir el código Ascii obtenido en el carácter correspondiente. Esto se logra utilizando la función `chr`:

```
letra_mayuscula := chr(valor_codigo_ASCII).
```


El siguiente programa resuelve en Pascal el problema planteado.

Ejemplo 3.15

Precondición:
- Lee un carácter que representa una letra minúscula

Poscondición:
- Se presenta en pantalla la misma letra leída, en mayúscula

```
Program convertir;
  var letra: char;          { letra a convertir en minuscula }
      distancia: integer;   { distancia ASCII entre dos letras }

  begin
    distancia := ord('a') - ord('A');
    writeln( 'Ingrese una Letra: ');
    readln( letra );
    if ( letra >= 'a' ) and ( letra <= 'z' )
      then writeln( chr( ord( letra ) - distancia ) )
      else writeln( 'El valor leido no es una letra minuscula' );
  end.
```

3.4 Tipos ordinales

Tres de los cuatro tipos de datos vistos hasta ahora tienen sus elementos **ordenados discretamente.**

Definición

Un tipo de dato se dice ordenado discretamente si para cada elemento que es parte del tipo existe un elemento anterior y otro posterior.

Por esta razón los tipos de datos enteros, carácter y lógico se denominan **tipos de datos ordinales.** Nótese que se excluye el tipo de datos real.

Cada elemento de tipo de dato entero, carácter o lógico tiene un elemento anterior y posterior. Por ejemplo, dado el número 4, su anterior es 3 y su posterior es 5; para el carácter 'c', su anterior es 'b' y su posterior es 'd' (utiliza para ello el código Ascii); en tanto que para los valores lógicos el anterior de *true* es *false*, y viceversa.

Para un valor de tipo de dato real no se puede definir un anterior o posterior. Se sabe que entre dos números reales siempre existe otro número real, por lo tanto, no se puede definir ni un anterior ni posterior a un valor dado.

Un detalle importante es que, en Pascal, cualquier variable de tipo ordinal (no solo de tipo entero) puede ser utilizada como variable de control en una estructura de control de tipo `for` (véase el capítulo 2).

3.4.1 Funciones sobre tipos de datos ordinales

Básicamente, existen tres funciones aplicables a los tipos de datos ordinales; una de ellas es la función `ord` descrita anteriormente, que además de aceptar como argumento un valor de tipo de dato carácter, puede recibir como parámetro de entrada un valor de tipo de dato entero y lógico.

Las funciones aplicables se presentan en la Tabla 3.5 en la que se describe, además de la sintaxis, la semántica asociada.

Función (sintaxis)	Descripción	Tipo de dato argumento	Tipo de dato resultado
`pred(x)`	Sucesor del elemento x	int, char o boolean	int, char o boolean
`succ(x)`	Predecesor del elemento x	int, char o boolean	int, char o boolean
`ord(x)`	Posición del elemento x dentro de los elementos del tipo	int, char o boolean	int

Tabla 3.5

Para el tipo de datos enteros:

```
succ(x) es igual a x+1, salvo el caso en el cual x sea MAXINT,
pred(x) es igual a x-1, salvo el caso en el cual x sea -MAXINT, y
ord(x) es igual a x.
```

Para los booleanos, el orden definido es primero *false* y luego *true*. De aquí que:

```
ord(false) =  0 y
ord(true) = 1.
```

Para los elementos que corresponden al tipo de datos carácter, el orden es el correspondiente al código ASCII. Esto es, el `succ` o `pred` de un elemento de tipo carácter es el elemento anterior o siguiente definido por dicho código. Esto es:

```
succ('v') =  'w',
pred('b') = 'a'.
```


En tanto que el `ord` de un elemento de tipo de dato carácter está determinado por el valor en código ASCII y acotado en el rango (0, 255).

Ejemplo 3.16

```
pred('H')  = 'G'
Succ(2) = 3
Ord(0) = 0
succ(false) = true
pred(110) = 109
succ('Z') = '['
```

3.5 Tipos de datos

Hasta aquí se han presentado los tipos de datos simples que se pueden considerar como estándar en la mayoría de los lenguajes de programación. Esto significa que el conjunto de valores que pueden tomar las variables de ese tipo y las operaciones que se pueden efectuar están predefinidas y acotadas por el lenguaje de programación. Dentro de los tipos de datos estándar, además de los simples (sin estructura), existen los denominados tipos compuestos (estructurados), que se describirán a partir del capítulo 5.

Sin embargo, la representación de los fenómenos reales que se tratan de resolver utilizando computadoras requiere normalmente la representación de elementos que no son tipos estándar de un lenguaje dado (por ejemplo, las calificaciones de exámenes son números entre 0 y 10; los días hábiles de la semana son lunes, martes, miércoles, jueves y viernes; un libro puede abstraerse como capítulos con páginas, cada una de ellas con líneas, cada línea con palabras y cada palabra con letras) y sobre los cuales las operaciones permitidas pueden ser específicas en cada caso (por ejemplo, obtener un promedio de exámenes aprobados; buscar el día hábil siguiente a una fecha; ubicar el capítulo de un libro donde aparece una frase; etc).

Entonces, un aspecto muy importante en la valoración que se haga de un lenguaje de programación es la posibilidad que brinde el mismo de representar datos no estándares, especificando sus valores permitidos y las operaciones válidas sobre los mismos. Cuando la incorporación de nuevos tipos de datos es transparente al usuario (es decir, que una vez que se agregó un tipo y sus operaciones el mismo se utiliza y valida exactamente igual que un tipo de dato estándar) se tiene un lenguaje con:

- Mejores posibilidades de abstracción de datos. Esto agrega ventajas evidentes como aumentar la riqueza expresiva del lenguaje con mucha claridad para la lectura de los programas.
- Mayor seguridad respecto de las operaciones que se llevan a cabo sobre cada clase de datos, permitiendo mayor número de validaciones.

- Límites preestablecidos sobre los valores posibles que pueden tomar las variables que corresponden al elemento del mundo real asociado.

El concepto de **tipos abstractos de datos**, una consecuencia de lo enunciado anteriormente, se desarrollará ampliamente en el capítulo 10.

Definición

Un tipo de dato definido por el usuario es aquel que no existe en la definición del lenguaje, donde el usuario es el encargado de determinar su denominación, y el conjunto de valores y operaciones que dispondrá el mismo.

3.5.1 Declaración de tipos

Los lenguajes de programación, en general, y Pascal en particular, permiten asignar nombres a los tipos y usarlos en declaraciones posteriores. La declaración de tipos presenta algunas ventajas:

- Flexibilidad: en caso de que sea necesario modificar la forma en que se representa la información solo se debe modificar una declaración, en lugar de una serie de declaraciones de variables.
- Documentación: se pueden asignar como nombres de tipo identificadores que representen la manera en que deben ser usados y no cómo están construidos, facilitando de esta manera el entendimiento y la lectura del programa.
- Seguridad: a través de la declaración de tipos se reducen los errores de correspondencia entre el valor que se pretende asignar a una variable y el previamente declarado. De esta manera se pueden obtener programas más confiables pues se realizan más chequeos sobre el código.

Cada declaración de tipos define **identificadores** que pueden ser utilizados en declaraciones de tipos, variables, procedimientos y funciones posteriores. El nuevo tipo puede ser tan simple como el nombre de un tipo predefinido o puede contener la descripción de un tipo totalmente nuevo.

En Pascal, los tipos deben ser declarados antes de ser utilizados. Siguiendo su notación, la declaración se realiza mediante la palabra clave `type`, de la forma:

```
type nombre_del_tipo = tipo_base
```

donde `nombre_del_tipo` es el identificador que representa el nombre con que se conocerá al tipo en el programa. Por otra parte, `tipo_base` puede ser un tipo de dato predefinido o alguno de los tipos de datos que se describirán en secciones y capítulos posteriores.

3.5.1.1 Renombrado de tipos

La declaración de tipos más simple es la que permite redefinir los tipos simples predefinidos en el lenguaje.

Ejemplo 3.17

```
type tipo_enteros = integer;
     tipo_reales = real;
     lógico = boolean;
     símbolos = char;
```

Estas formas de declaraciones introducen nuevos tipos que pueden resultar útiles, por ejemplo, para escribir procedimientos de propósito general.

Ejemplo:

```
type tipo_numerico = integer;
```

Una vez declarado el tipo, puede utilizarse para definir variables, como por ejemplo:

```
var a : tipo_numerico
```

El siguiente programa presenta la definición de una función que retorna el valor menor entre dos números.

Código

```
Function minimo( x, y: tipo_numerico ): tipo_numerico;
  { retorna el mínimo entre x e y }
  begin
    if x < y
      then minimo := x
      else minimo := y;
  end.
```

Nótese que una vez escrita esta función, la decisión del tipo al que hace referencia `tipo_numerico` se demora hasta que la función se inserte dentro de un programa. Por ejemplo, `tipo_numerico` podría ser el definido anteriormente (`integer`) o podría ser de la forma

```
tipo_numerico = real;
```

Dependiendo de cual sea la definición que se efectuó, la función retornará el mínimo entre dos enteros o entre dos reales. Este manejo de tipos permite unificar fácilmente los proyectos en equipo.

Ejemplo 3.18

Si otro programador tiene definida la función presentada en programa anterior, se puede salvar la diferencia de nombres codificando

```
type numero = tipo_numerico
```

```
Function maximo( x, y: numero ): numero;
  { retorna el máximo entre x e y }
  begin
    if x > y
      then maximo := x
      else maximo := y;
  end.
```

3.5.2 Tipos, asignaciones y subprogramas

Para que una asignación sea válida, los tipos de la variable y la expresión deben coincidir. Además, existen reglas estrictas respecto al pasaje de parámetros por referencia: el tipo de un parámetro actual (o argumento) debe coincidir con el tipo del parámetro formal.

Dos variables tienen el mismo tipo de dato si son definidas de acuerdo con el siguiente esquema:

a. Se definen en la misma declaración.
b. Utilizan mismo identificador de tipo.
c. Utilizan identificadores sinónimos.

Ejemplo 3.19

Sean las siguientes declaraciones:

```
type tipo_numerico = integer;
     numero = integer;
```

Las variables declaradas a continuación son del mismo tipo por alguno de los motivos expuestos:

```
var a, b : tipo_numerico;          por a)
var c : tipo_numerico;
    d : tipo_numerico;             por b)
var e : tipo_numerico;
    f : numero;                    por c)
```


3.6 Tipos de datos definidos por el usuario

3.6.1 Tipo de dato enumerativo

En los primeros lenguajes de programación, todo debía expresarse en binario, como ceros y unos. A medida que estos lenguajes evolucionaron el vocabulario de descripción se expandió. Sin embargo, un conjunto limitado de medidas básicas (usualmente equivalentes a los tipos de datos reales, enteros, lógicos y caracter) debían emplearse para describir cualquier valor imaginable.

Pero el mundo real está lleno de objetos que tienen sus propios nombres. Por ejemplo: este mes podría ser enero o septiembre; mis programas favoritos podrían ser deportivos, documentales, periodísticos o películas; en una verdulería podría elegir entre comprar manzanas, peras, ciruelas, tomates o rabanitos; etc.

Algunos lenguajes, como Pascal, permiten que tales grupos de valores nombrados constituyan la base de un **tipo de dato enumerativo** que es definido por el usuario.

3.6.1.1 Declaración

En general los lenguajes de programación, como Pascal, para declarar un tipo enumerativo, solo se deben enumerar sus valores encerrándolos entre paréntesis, de la forma:

```
type nombre_tipo_enum = (valor_1, valor_2, ..., valor_n)
```

Una vez definido el tipo se pueden definir variables del mismo.

Ejemplo 3.20

```
type colores = (azul, blanco, negro, verde);
     frutas = (manzana, pera, ciruela, banana, sandia);
     verduras = (lechuga, rabanito, remolacha, zanahoria);
     dias_de_la_semana = (domingo, lunes, martes, miercoles, jueves,
                          viernes, sabado);
var color : colores;
    postre : frutas;
    verdura_1, verdura_2 : verduras;
    dia : dias_de_la_semana;
```

Los identificadores que detallan los valores posibles del tipo son **constantes simbólicas**. Pascal no permite que una constante simbólica aparezca en la definición de dos tipos enumerativos diferentes. Otros lenguajes, como ADA, no presentan esta restricción y, por lo tanto, aceptan que una misma constante simbólica figure en la definición de más de un tipo de datos.

Ejemplo 3.21

```
type ciudad_1 = (LaPlata, Cordoba, Mendoza, Rosario);
     ciudad_2 = (MarDelPlata, Mendoza, Neuquen);{ILEGAL !+ se repite
                                                 Mendoza como constante}
     no_valido = ('a','x','P','Z'); {ILEGAL !! pues los valores son char}
```

3.6.1.2 Operaciones sobre datos enumerativos

Las operaciones válidas sobre las variables de tipo de dato enumerativo son reducidas para la mayoría de los lenguajes que la soportan, Pascal no escapa a este hecho, solo se admiten operaciones de asignación o de comparación. No es posible efectuar operaciones de entrada-salida sobre un valor de tipo enumerativo, debido a esta causa, las variables de este tipo son de uso interno, generalmente para brindar claridad al programa. Otros lenguajes como ADA, permiten en determinados casos, efectuar estas operaciones de entrada-salida.

Ejemplo 3.22

Sean las siguientes declaraciones:

```
type colores = (azul, blanco, negro, verde);
var   color: colores;
```

Entonces, son válidas las siguientes operaciones:

```
color := azul;                                  {asignación}
if (color = blanco) or (color = negro) then...  {comparación}
for color := azul to verde do...      {color es de tipo ordinal}
```

No son válidas operaciones tales como:

```
read (color);
write (color);
```

Ejemplo:

Sea el siguiente problema. Se lee una secuencia de caracteres conteniendo una expresión matemática del tipo:

```
3 + { 5 - a *  b } - (c / 4 ) + [ c *  b + 1 ].
```

Se debe realizar un algoritmo que determine si la expresión está balanceada, es decir, si cada signo de apertura (paréntesis, corchete, llave) tiene su correspondiente signo de cierre. Se considera, para el algoritmo, que luego de un signo de apertura se recibe uno de cierre, no que puede llegar otro de apertura.

El siguiente programa presenta la solución:

Código

Precondición:

- La expresión no posee signos de apertura anidados

Poscondición:

- Se determina si la expresión está o no balanceada.


```
Program balanceo;
  type estados_posibles = (normal, parentesis, corchete, llave, desbalanceado );
  var  estado: estados_posibles;
       c: char;
  begin
    estado := normal;
    read( c );
    {se analiza cada caracter hasta terminar o encontrar que se desbalancea
     la secuencia de entrada}
    while( c <> '.' ) and ( estado <> desbalanceado ) do
      begin
        case estado of
          normal: case c of     {cuando el estado es normal, se aceptan
                                 solamente símbolos de apertura y cambia
                                 el estado}
                    '(': estado := parentesis;
                    '[': estado := corchete;
                    '{': estado := llave;
                    ')',']','}': estado := desbalanceado; {se encuentra
                                   un símbolo de cierre, estado erroneo}
                  end;
          parentesis: case c of {se acepta como valido un carácter de
                                 cierre de parentesis, otra símbolo es
                                 erroneo}
                    ')': estado := normal;
                    '(','{','[',']','}': estado := desbalanceado;
                  end;
          corchete: case c of   {se acepta como valido un caracter de
                                 cierre de corchete, otra simbolo es
                                 erroneo }
                    ']': estado := normal;
                    '(','{','[',')','}': estado := desbalanceado;
                  end;
          llave: case c of      {se acepta como valido un caracter de
                                 cierre de llave, otra simbolo es erroneo}
                    '}': estado := normal;
                    '(','{','[',']',')': estado := desbalanceado;
                  end;
        end;
        read( c );
      end;
    if estado = normal
      then writeln( 'Expresion balanceada' )
      else writeln( 'Expresion desbalanceada' );
  end.
```

3.6.1.3 Funciones predefinidas sobre datos enumerativos

En Pascal, las funciones predefinidas sobre el tipo de dato enumerativo son las que se describieron para los datos de tipo ordinal (`ord`, `succ`, `pred`).

Ejemplo 3.23

Sean las siguientes declaraciones:

```
type colores = (azul, blanco, negro, verde);
 dias = (domingo, lunes, martes, miercoles, jueves, viernes, sabado);
```

Luego:

```
ord(azul) = 0
pred(azul) = Inválido
succ(azul) = Blanco
ord(martes) = 2
succ(sabado) = Inválido
pred(sabado) = viernes
```

3.6.2 Tipo de dato subrango

En algunas ocasiones no es necesario utilizar todos los valores que provee un tipo ordinal, sino solo una parte de los mismos. Es en estos casos que se necesita trabajar con un subrango de los tipos definidos. Por ejemplo, un problema en el cual se necesitan solo los números naturales, o solo las letras minúsculas, o solo los días laborables de la semana.

Definición

Un tipo de dato subrango es un tipo ordinal, que consiste de una secuencia contigua de valores de algún tipo ordinal (llamado tipo base del subrango).

En general, el subrango se determina mediante los límites inferior y superior de la secuencia, separados por dos puntos, de la forma:

```
type nombre_subrango = limite_inferior..límite_superior
```

Este tipo de datos permite que el lenguaje evalúe si los valores que son asignados se encuentran dentro del rango establecido.

Ejemplo 3.24

```
type naturales = 0 .. MAXINT;                {Naturales, subrango de integer}
     minusculas = 'a' .. 'z';                {Minúsculas, subrango de char}
     dias = (domingo, lunes, martes, miercoles, jueves, viernes, sabado);
     dias_laborables = lunes .. viernes;     {Días laborables, subrango de}
                                             {días}
     algunos_reales = 1.0 .. 10.5            {ILEGAL !! El tipo base no}
                                             {es ordinal}
```

3.6.2.1 Operaciones y funciones predefinidas sobre datos subrango

Las operaciones y las funciones predefinidas que se pueden realizar sobre un dato de tipo subrango son las mismas que las de su tipo base.

3.6.3 Tipo de dato conjunto

Algunos lenguajes de programación permiten trabajar con conjuntos. La noción de conjuntos es la misma que la ya conocida en Matemáticas: es una colección homogénea de elementos, sin repetición, sin relación de orden entre ellos, e ilimitada.

Definición

Un conjunto, desde el punto de vista informático, es una colección de datos simples, todos del mismo tipo.

De esta definición se desprende que se pueden tener conjuntos de enteros, caracteres o enumerativos. En Pascal, además, el número máximo de elementos por cuestiones de implementación, está acotado a 256.

3.6.3.1 Declaración de conjuntos

El tipo conjunto se declara de la forma:

```
type conjunto = set of tipo_ordinal
```

donde `tipo_ordinal` es el tipo al cual pertenecen los elementos del conjunto y al que se lo denomina tipo base. Una vez definido el tipo conjunto, podemos definir una variable conjunto de la siguiente manera:

```
var nombre_variable: conjunto;
```

También es posible declarar directamente una variable como un conjunto sin una declaración previa de tipo:

```
var nombre_variable: set of tipo_ordinal;
```

Ejemplo 3.25

```
type  dias = (domingo, lunes, martes, miercoles, jueves, viernes, sabado);
      uno_al_100 = 1 .. 100;
      letras = set of char;
      algunos_dias = set of dias;
      algunas_frutas = set of (manzana, pera, ciruela, banana, sandia);
      algunos_numeros = set of uno_al_100;
```

3.6.3.2 Construcción de un conjunto

Los conjuntos pueden ser asignados a variables de tipo conjunto. También pueden usarse como operandos en ciertos tipos de expresiones lógicas.

El conjunto se construye definiendo los elementos individuales consecutivamente, encerrados entre corchetes y separados por comas. Así un conjunto individual aparecerá como

```
[ elem.1, elem.2, ..., elem.M ]
```

Ejemplo 3.26

```
Program define_conjunto;
  var letras: set of char;
  begin
    letras := ['a','b','c','x','y','z']
    .....
  end.
```

los elementos incluídos se definen como miembros del conjunto. En el ejemplo, las letras 'a','b','c','x' y 'z' son los miembros del conjunto letras.

El conjunto que no contiene elementos se denomina conjunto **vacío** o **nulo** y se denota por un corchete de apertura seguido de un corchete de cierre ([ ]).

Algunos de los miembros del conjunto pueden representarse como variables, siempre que las mismas representen elementos del tipo base apropiado. Además, si alguno de los miembros del conjunto son elementos consecutivos del tipo base puede representarse como un subrango, de la siguiente forma:

```
[primer_elem_consecutivo .. ultimo_elem_consecutivo]
```


Ejemplo 3.27

```
Program define_conjunto_2;
  type let = set of char;
  var letras_2: let;
  begin
    letras_2 := ['a'..'t']
    ......
  end.
```

Con lo cual el conjunto `letras_2` incluiría todas las letras minúsculas desde la 'a' hasta la 't'.

En caso de especificar los valores de un conjunto como un subrango donde los valores están en orden inverso (por ejemplo `['t'..'a']`), el conjunto se considera vacío.

Tal como expresa la definición, un conjunto no puede tener elementos repetidos. Sin embargo, es posible especificar un elemento una o más veces, en cuyo caso las repeticiones serán ignoradas y se considerará una sola aparición.

3.6.3.3 Operaciones sobre datos de tipo conjunto

Las operaciones válidas sobre conjuntos dependen de cada lenguaje, en Pascal, además de la asignación, son las tradicionales de unión, intersección, diferencia y evaluación de pertenencia. La Tabla 3.7 presenta dichas operaciones

Operación	Operador	Tipo de resultado
unión	+	conjunto
Intersección	*	conjunto
Diferencia	-	conjunto
Pertenencia	in	lógico

Tabla 3.7

Ejemplo 3.28

Sea la siguiente declaración.

```
type conjunto = set of char;
var conj_1,
    conj_2,
    conj_3 : conjunto;
```

las siguientes operaciones son válidas:

```
conj_1 := [ ];                      conj_1 contiene el conjunto vacío
conj_2 := [ 'a' ];                  conj_2 contiene la letra a.
conj_2 := conj_2 + ['b' .. 'h'];    conj_2 contiene las letras desde a hasta h
conj_1 := [ 'c', 'f' ]              conj_1 contiene las letras c y f
conj_3 := conj_1 * conj_2           conj_3 contiene las letras c y f
conj_3 := conj_3 + conj_1           conj_3 contiene las letras c y f
conj_1 := conj_3 - conj_2           conj_1 tiene el conjunto vacío.
( 'd' in conj_2 )                   da como resultado true
```

Los operadores relacionales que pueden utilizarse con conjuntos son: <>, <=, >=, =. Si A y B son variables del mismo tipo conjunto:

- A <> B : retorna verdadero si los conj.son distintos.
- A <= B : retorna verdadero si A está incluído en B
- A >= B : retorna verdadero si B está incluído en A
- A = B : retorna verdadero si A y B son iguales.

Código

Sea el siguiente problema. Escribir un programa que construya un conjunto con una sucesión de números enteros que se obtienen desde teclado. Dicha sucesión termina con cero y los valores ingresados pertenecen al intervalo [20,50].

```
Program ejemplo_conjunto;
  type rango = 20..50;
  var conjunto: set of rango;
      entero: rango;
  begin
    readln( entero );
    conjunto := [];
    {para cada elemento ..... }
    while ( entero > 0 ) do
      begin
        conjunto := conjunto + [entero]; {agrego el numero al conjunto}
        readln( entero );
      end;
  end.
```

Ejemplo 3.29

El programa presenta un procedimiento que reciba dos conjuntos del mismo tipo como parámetros y retorne un nuevo conjunto que representa la intersección de los dos conjuntos recibidos

```
procedure interseccion( a,b: conjunto; var c: conjunto );
  begin
    c := a * b;
  end;
```


Ejemplo 3.30

Realizar un programa en el que, dada una secuencia de caracteres terminada en punto, determine cuáles son las letras que pertenecen a la palabra Pascal.

Precondición:

- Se leen caracteres que terminan con un punto

Poscondición:

- Se informan los caracteres leídos de la palabra pascal

```
program conjuntos;
  var pascal: set of char;
      letra: char;
  {dado un caracter cualquiera, solo a las letras las convierte a mayúscula}
  procedure mayuscula(var caracter: char);
    begin
      if caracter in ['a'..'z']
        then caracter:= chr(ord(caracter) - ord('a')+ord('A'));
    end;

  {cuerpo del principal}
  begin
    pascal := ['P','A','S','C','L'];
    readln(letra);
    while letra <> '.' do
      begin
        mayuscula(letra);        {pasa la letra leida a mayúscula}
        if letra in pascal       {si pertenece al conjunto ....}
          then writeln(letra, ' pertenece a la palabra PASCAL');
        readln(letra);
      end;
  end.
```

Un dato interesante para mencionar, es que, al igual que las variables de tipo enumerativo, no es posible realizar operaciones de lectura y escritura sobre variables de tipo conjunto.

3.6.4 Tipo de dato string

En la mayoría de los lenguajes de programación existe un tipo estándar que es la hilera de caracteres (*string*).

De acuerdo a definiciones previas, un carácter es una representación de una letra, número o símbolo que se guarda internamente de acuerdo a su representación determinada por el código ASCII. Cuando se trabaja con el tipo de dato *string*, se tienen n caracteres tratados como una única variable, donde n proviene de la definición del *string*, como se verá más adelante.

Definición

Un tipo de dato *string* es una sucesión de caracteres que se almacenan en un área contigua de la memoria y que puede ser leído o escrito.

El tipo de dato *string* representa un dato de tamaño dado que resulta de la concatenación de los caracteres que lo forman.

3.6.4.1 Declaración de *string*

El tipo de dato *string* debe indicar el tamaño máximo que se desea manejar, y se define como:

```
type nombre_de_string = string[ longitud ];
```

donde `nombre_de_string` es el identificador del tipo, y `longitud` es el número máximo de caracteres que puede contener.

Ya definido el tipo de dato *string* se pueden declarar variables de ese tipo de la siguiente forma:

```
var nombre_de_variable_string: nombre_de_string;
```

Además, es posible declarar variables de tipo de dato *string* como se indica a continuación:

```
var nombre_de_variable_string: string[ longitud ];
```

Ejemplo 3.31

```
type hilera1 = string[10];
     hilera2 = string[25];
var  h1, h2, h3: hilera1;
     h4: hilera2;
     h5, h6: string [14];
     h7: string;
```

Nótese que la última definición (h7) es un caso particular para el lenguaje Pascal, debido a que no se indica explícitamente la cantidad de caracteres que forman el *string*. En este caso, la variable h7 podrá contener como máximo 255 caracteres debido a la implementación. Existen otros lenguajes, por ejemplo C, en donde esta restricción no existe (Leestma, 1984).

Para asignar valor a una variable de tipo de dato *string* se procede igual que lo visto para las variables de tipo de dato carácter.

Ejemplo 3.32

Las siguientes asignaciones son válidas:

```
h1 := 'pepe';
h2 := '678#$@abc';
h3 := '1234567890abcd';
h4 := '';                    {dos apóstrofes denotan string nulo}
```


La asignación efectuada sobre h3 tiene una particularidad. Si se observa cuidadosamente, la variable h3, que es de tipo hilera1, está definida como un *string* de tamaño 10. La asignación efectuada es de 14 símbolos, en este caso y por el tamaño declarado, la variable h3 solo puede contener 10 símbolos, por lo tanto h3 recibe los primeros 10 caracteres (los dígitos), quedando descartados los restantes (las letras).

3.6.4.2 Operaciones sobre datos de tipo *string*

Las operaciones válidas sobre *string* son la asignación y la denominada concatenación. Esta operación permite adosar un *string* a continuación de otro.

Ejemplo 3.33

Sea la siguiente declaración.

```
type cadena = string[30];
var st1, st2: cadena;
```

las siguientes operaciones son válidas:

```
st1 := '';                    st1 contiene el string nulo
st2 := ' ';                   st2 contiene un blanco
st1 := '123abc'
st2 := '345'
st2 := st2 + st1              st2 contiene '345123abc'
st1 := ' ' + st1              st1 contiene ' 123abc'
st3 := st2 + ' ' + st1        st3 contiene '345123abc  123abc'.
```

Nótese la diferencia que existe entre las dos primeras asignaciones; `st1` recibe como asignación un *string* nulo (o sea nada), mientras que `st2` recibe un blanco. La última asignación contiene dos espacios en blanco; el primero, dado que se concatena un blanco entre `st2` y `st1`, y el segundo debido a que, por la asignación anterior, `st1` tiene un blanco como primer símbolo.

3.6.4.3 Operadores lógicos sobre datos de tipo *string*

Los operadores relacionales que pueden utilizarse con *string* son: =, <>, >, <, <=, >=, =. Las comparaciones entre variables de tipo de dato *string* merecen una consideración especial.

Los operadores relacionales pueden aplicarse entre dos variables de tipo *string* sin tener necesidad de que las mismas sean de igual longitud. El resultado se determina de acuerdo al siguiente criterio:

- Operador = : si las cadenas que se comparan son de igual longitud y contienen los mismos símbolos, en el mismo orden, el resultado es verdadero. Ahora bien, si tienen distinta longitud y se le ha asignado distinta cantidad de símbolos, la comparación es falsa.

- Operador <> : es el caso opuesto al planteado anteriormente.
- Operadores >, <, >=, <= : cada uno de ellos actúa comparando símbolo a símbolo de acuerdo a su aparición en el código ASCII, hasta poder determinar un resultado para el operador. De esta forma es perfectamente lícito comparar el string `'pepe'` contra `'pipo'`; el resultado, en este caso, será que `'pipo'` es mayor que `'pepe'`, debido a que la letra i aparece luego que la e.

Cabe acotar que esta forma de resolver las comparaciones es válida para algunas versiones de Pascal, pero en otros lenguajes la resolución de los operadores (en particular = y <>) puede variar.

Ejemplo 3.34

Sea la siguiente declaración:

```
var a: string[4];
        b: string[3];
```

Si asignamos a estas variables los valores:

```
a := 'abcd';
b := 'abc';
```

Luego:

```
a = b  =>    false
a <> b =>    true
a > b  =>    true
a < b  =>    false
a >= b =>    true
a <= b =>    false
```

Si, ahora, se asignan los valores:

```
a := 'abc';
b := 'abc';
```

Luego:

```
a = b  =>    true
a <> b =>    false
a > b  =>    false
a < b  =>    false
a >= b =>    true
a <= b =>    true
```

Existe un conjunto de funciones predefinidas para operar sobre tipos de datos *string*. La tabla 3.8 resume las principales funciones, la sintaxis utilizada es la definida por el lenguaje Pascal:

Función (sintaxis)	Descripción	Tipo de dato argumento	Tipo de dato resultado
Copy (string, posición, cantidad)	Retorna una porción de string que comienza en posición y de largo cantidad	varios	string
Length (string)	devuelve la longitud del string (cantidad de caracteres distinto de nulo	string	integer
Pos (string_1, String_2)	retorna la posición de comienzo del string_1 dentro del string_2 (si string_1 no existe retorna 0)	string	integer
Concat (string1, String2,....)	retorna la concatenación de todos los string's que se pasan como parámetro	string	string

Tabla 3.8

Ejemplo 3.35

Sean las siguientes variables, todas de tipo de dato *string* de longitud n.

```
s1 := 'abcdef'
s2 := 'de'
s3 := 'fe'
```

Luego:

```
length ( s1 )       =>   6
length( s2 )        =>   2
copy( s1, 2, 3)     =>   'bcd'
copy( s1, 4 )       =>   'def'
concat(s2, s3)      =>   'defe'
length(s1+s2)       =>   8
pos( 'bc', s1)      =>   2
pos( s2, s1 )       =>   4
pos( s3, s1 )       =>   0
```

Algunos comentarios: En el cuarto caso, a la función copy no se le define el tercer parámetro (la cantidad de símbolos que retorna) en este caso se retorna desde el valor que indica el segundo parámetro y hasta el último símbolo definido.

Ejemplo 3.36

Se lee un conjunto de palabras terminadas en blanco. Realizar un algoritmo que determine e informe la longitud de cada palabra.

```
program longitud;
  type str30 = string[ 30 ];
  var palabra: str30;
  begin
    readln( palabra );
    while( palabra <> '' ) do
      begin
        writeln( 'la longitud de la palabra es:', length(palabra) );
        readln( palabra );
      end;
  end.
```

Ejemplo 3.37

Se lee una sucesión de 10 palabras y se pide informar para cada una de ellas la cantidad de vocales que contiene y, además, la cantidad total de letras 'a' encontradas.

Precondición
- Se leen 10 palabras

Poscondición
- Se informa lo deseado por palabra y en general

```
program longitud;
  type str30 = string[ 30 ];
  var palabra: str30;
      cant,
      cantidad_vocales,
      cantidad_a: integer;

  procedure contar( pal: str30; var vocal, a: integer );
    var posicion: integer;
        conjunto: set of char;
    begin
      conjunto := ['a','e','i','o','u'];
      vocal := 0; {para cada palabra el contador de vocales empieza en 0}
      for posicion := 1 to length( pal ) do
        begin
          if( copy( pal, posicion, 1 ) in conjunto ) {se controla
                                                      que sea vocal}
            then begin
              vocal := vocal + 1;
```



```
              if( copy( pal, posicion, 1 ) = 'a' ) {se controla que sea una a}
                then a := a + 1
            end;
        end;
    end;

  begin
    cantidad_a := 0;
    for cant := 1 to 10 do
      begin
        readln( palabra );  { lee la palabra }
        contar( palabra, cantidad_vocales, cantidad_a ); {cuenta vocales}
        writeln( 'La cantidad de vocales es: ', cantidad_vocales );
      end;
    writeln( 'La cantidad total de letras "a" es: ', cantidad_a );
  end.
```

Conclusiones

Se han presentado los tipos de datos simples que aparecen como estándar en la mayoría de los lenguajes de programación.

Si bien la ejemplificación de operaciones está centrada en Pascal, el lector podrá aplicar los conceptos con variantes menores a una amplia gama de lenguajes.

El concepto de tipo es esencial en la programación de algoritmos y se profundizará su tratamiento en los capítulos que siguen del texto.

Ejercicios

1. Sean las siguientes variables: a, b de tipo entero; c, d de tipo real; e,f de tipo carácter; g de tipo lógico. Indicar en cuáles de las siguientes asignaciones son válidas y cuáles no, para estos últimos casos indicar por qué no lo son.

a) a := 20	b) g := 12	c) f := '0'
d) b := 500	e) c := 0	f) d := c;
g) e := 'f'	h) e := f	i) a := 12.56
j) g := true	k) f := g	l) b := 0.45

2. a) Desarrolle una función que lea una secuencia de caracteres terminada en punto y retorne la cantidad de caracteres leídos.
   b) ¿Qué debería modificar al ejercicio anterior para retornar, ahora, la cantidad de letras 'f' que encuentra.?

3. Dada la siguiente declaración de tipo:

```
Type mes=(ene,feb,mar,abr,may,jun,jul,ago,set,oct,nov,dic)
```

Determine, si es posible, el valor de cada una de las expresiones siguientes:

a) `ene < ago`	g) `succ( pred( mar ) )`
b) `set <= set`	h) `succ( pred( ene ) )`
c) `succ( set )`	i) `ord( ene )`
d) `pred( abril )`	j) `ord( set ) - ord( ene )`
e) `succ( succ( ago ) )`	k) `ord( succ(may )) - ord( may )`
f) `pred( pred( ago ) )`	l) `succ( pred( jul ) )`

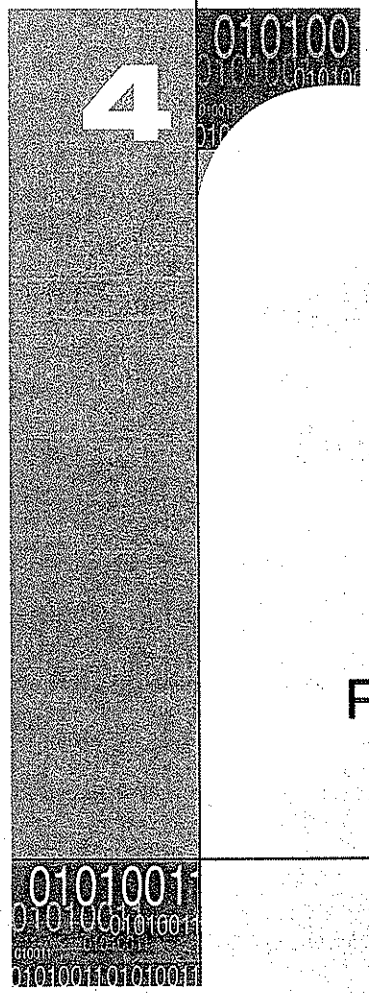
4. a) Desarrolle un programa que lea nombres de animales, e invoque a un procedimiento que devuelva el nombre del animal en una variable de tipo enumerativo. Los animales pueden ser: hormiga, perro, gato, pez, ballena y gallina. Analice el caso en que el nombre no corresponda a un animal.

   b) Realice un procedimiento que imprima el nombre del animal habiendo recibido un enumerativo que contiene dicho nombre.

5. Escriba un programa que lea una secuencia de caracteres terminada en punto e informe cuántas veces aparecen las letras de la palabra 'PROGRAMACION', sabiendo que las letras pueden ser tanto mayúsculas como minúsculas.

6. Desarrollar un módulo que reciba un número entero y que devuelva la cantidad de cifras pares que el mismo contiene.

7. Generalizar el algoritmo que lee nombres de ciudades, suponiendo, ahora, que dichos nombres pueden contener más de una palabra y se debe convertir la primera letra de cada una a mayúscula.



Capítulo 4

Procedimientos y funciones. Parámetros

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

4

Procedimientos y funciones. Parámetros

Objetivo

En este capítulo se profundizan los conceptos relacionados con la definición de módulos, tanto procedimientos como funciones. Se define y ejemplifica cuidadosamente la forma de comunicación entre ellos, a través de la utilización de parámetros.

La definición de parámetros se presenta en forma genérica, analizando las características que tienen los lenguajes de programación en general. Luego se discute la implementación estándar que tiene Pascal.

Este capítulo retoma los conceptos de modularización presentados previamente en los capítulos 1 y 2.

4.1 Subprogramas o módulos

En el capítulo 2 se definió el concepto de Descomposición de Problemas y Modularización como un aspecto fundamental para el desarrollo de algoritmos complejos. A partir de la descomposición funcional de un problema, utilizando por ejemplo un diseño descendente *(Top-Down)*, se logra dividir un problema en un conjunto de tareas cada vez más sencillas de resolver.

Por otra parte, el concepto de modularización puede asociarse al concepto de reuso. La reutilización consiste en aprovechar el trabajo realizado en desarrollos anteriores. De esta forma, no es necesario volver a implementar algunas soluciones que ya fueron suficientemente probadas. En la medida que se realicen y prueben módulos específicos, estos podrán ser utilizados posteriormente por otros algoritmos que lo necesiten.

Definición

Cada tarea recibe el nombre de subprograma, módulo o rutina. Cada una puede desarrollarse de manera independiente del resto.

Un módulo puede realizar las mismas acciones que un programa (leer un dato, realizar un cálculo, presentar un resultado, etc.). La diferencia está en que el módulo se ejecuta cada vez que el programa principal lo invoca. Además, puede ocurrir que un subprograma puede llamar o invocar a otros subprogramas. Se discutirán posteriormente algunos conceptos que profundizan este tema.

Cada módulo debe tener un nombre que lo identifique, además es posible que tenga una serie de parámetros asociados, como se explicará más adelante. El nombre del módulo es utilizado para la invocación del mismo.

Si un programa o subprograma necesita ejecutar las acciones que corresponden a algún módulo, en Pascal debe poner dentro de su código el nombre de la rutina como una instrucción más. Cuando se ejecute esta instrucción se "invocará" a esta rutina transfiriendo el control de ejecución a la misma. Una vez finalizada la última instrucción del módulo, el control retornará a la siguiente instrucción del programa o subprograma que lo llamó. Otros lenguajes de programación pueden tener una sintaxis diferente para invocar un módulo, por ejemplo, utilizar la palabra clave `call`. Figura 4.1.

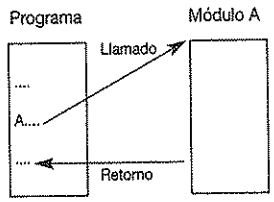


Figura 4.1

Los módulos o rutinas pueden ser funciones o procedimientos.

4.1.1 Alcance de un módulo

Anteriormente se indicó que un subprograma puede ser invocado por el programa principal o algún otro subprograma. En general, esta afirmación es válida, pero existe un conjunto de reglas que se deben cumplir para poder llevarla a cabo.

Estas reglas establecen el alcance o visibilidad de cada módulo, a partir del cual es posible o no que otro lo invoque. La definición del alcance de un módulo es dependiente del lenguaje, de aquí que se discutirá la regla que plantea Pascal, la cual no es, necesariamente, igual a otros lenguajes de programación.

La Figura 4.2 presenta un diseño *Top–Down* para un problema genérico. En él, un programa principal A, se descompone en tres tareas secundarias: B,C,H. A su vez, B se descompone en las tareas B y E, en tanto que c en los módulos F y G.

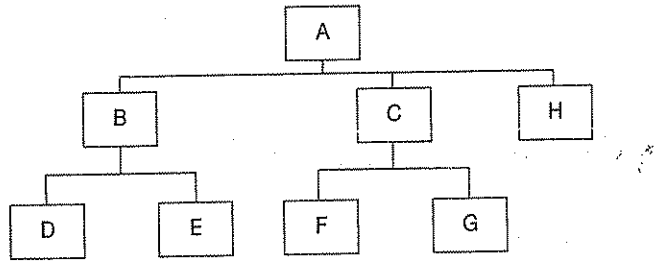


Figura 4.2

El siguiente programa presenta el esqueleto que representa al diseño de la figura 4.2, en él se ejemplifican las distintas posibilidades de invocación de los módulos y se agrega como comentario cuándo son correctas y cuándo no en Pascal.

Código

```
Program A;
  Procedure B;
    Procedure D;
      begin  { del procedimiento D }
        b;
        c;   { no es válido}
      end;
    Procedure E;
      begin  { del procedimiento E }
        b;
        c;   { no es válido }
        d;
      end;
```

```
    begin  { del procedimiento B }
      d;
      c;   { no es válido }
    end;
  Procedure C;
    Procedure F;
      begin { del procedimiento F }
        b;
        c;
        d; { no es válido }
        e; { no es válido }
        g: { no es válido }
        h; { no es válido }
      end;
    Procedure G;
      begin { del procedimiento G }
        b;
        c;
        d; { no es válido }
        e; { no es válido }
        f;
        h; { no es válido }
      end;
    begin { del procedimiento C }
      b;
      d; { no es válido }
      e; { no es válido }
      h; { no es válido }
    end;
  Procedure H;
    begin { del procedimiento H }
      b;
      c;
      d; { no es válido }
      e; { no es válido }
      f; { no es válido }
      g; { no es válido }
    end;
begin  { del programa principal A }
  b;
  c;
  d;  { no es válido }
  e;  { no es válido }
  f;  { no es válido }
  g;  { no es válido }
  h;
end.
```


Se puede observar que un módulo puede invocar a otro módulo que esté definido en su contexto. El contexto ("visibilidad") del módulo depende de su ubicación dentro de la definición de las rutinas.

Un módulo conoce, *a priori*, todo lo que esté definido anteriormente a él. De aquí que el módulo C pueda invocar al módulo B y no lo opuesto.

Un módulo conoce todo lo que está definido inmediatamente en su interior. De aquí que el programa principal puede llamar a B, C y H.

Como corolario de lo anterior, el programa principal **no** puede invocar a los módulos D, E, F, G, pues no están definidos como internos, sino que son subprogramas de otros (B y C en este caso).

4.2 Procedimientos

Definición

Un procedimiento es un conjunto de instrucciones que realizan una tarea específica y como resultado del cual puede retornar o no uno o más valores como respuesta.

Los casos en que un procedimiento no devuelve ningún valor pueden ser, por ejemplo, el de rutinas que se limitan a imprimir un título.

Un procedimiento o proceso se identifica mediante un nombre y, eventualmente, está seguido de una lista de parámetros. El programa 4.2 presenta el esqueleto de la definición de un procedimiento. Cuando se invoca al procedimiento se escribe como una instrucción el nombre del mismo y, entre paréntesis, se indica la lista de parámetros.

Código

```
Program ejemplo;
  Procedure nombre_del_proceso ( lista_de_parametros );
    type ...
    var ....
  begin
 ....
  end;
begin    {del programa ejemplo }
  ...
  nombre_del_proceso( lista_de_parametros );
  ...
end.
```

4.3 Funciones

Tal como se ha estudiado en Matemáticas, una función es una operación que, a partir de uno o más valores, produce otro valor denominado **resultado**.

Ejemplo 4.1

```
F(a) = a + 4
F(x,y) = x² * y/2
```

Definición

Una función es un módulo que realiza una tarea específica y que como resultado de ella retorna un único valor como respuesta.

Analizando la definición de procedimiento y función, el lector notará una gran similitud entre ambas. La diferencia radica básicamente en que las funciones retornan siempre un único resultado, en tanto que los procedimientos pueden retornar 0, 1 o más resultados.

La segunda diferencia radica en la forma en que se define una función. A diferencia de un procedimiento, la función indica en su definición el tipo de dato que retornará como resultado. A continuación se muestra un ejemplo genérico de definición e invocación de una función. Nótese que la función produce un resultado el cual debe ser "consumido" por el programa o subprograma que la invoca. Consumir el resultado puede significar: asignarlo a una variable del mismo tipo, imprimirlo o usarlo como parte en una condición lógica.

Código

```
Program ejemplo;
  var resultado: tipo_de_dato;
  Function nombre_de_la_funcion ( lista_de_parametros ): tipo_de_dato;
    type ...
    var ....
  begin
    ....
  end;
begin    {del programa ejemplo }
  ...
  resultado := nombre_de_la_funcion( lista_de_parametros );
  ...
end.
```


4.4 Parámetros

Generalmente, los módulos, tanto procedimientos como funciones, necesitan para operar algunos datos que están disponibles en el programa o subprograma que lo invoca. En ambos casos es necesario "comunicar" módulos. La forma en que se realiza esta comunicación es a través de parámetros.

Definición

Se denomina parámetros a la serie de datos con los que se comunican los módulos.

Los parámetros de un módulo deben definirse, como se indicó anteriormente, en el encabezado del mismo. Cada parámetro debe especificar el tipo de datos con el que se corresponde. En la invocación del módulo deben definirse también estos parámetros. Los parámetros que se definen en el llamado del módulo reciben el nombre de **parámetros actuales**, en tanto que los parámetros descritos en el encabezado del módulo invocado (procedimiento o función) se denominan **parámetros formales**.

La mayoría de los lenguajes de programación exigen que el número de parámetros actuales y formales sea el mismo y de igual tipo. Pascal chequea esta condición que, en caso de no cumplirse, impide ejecutar el programa. En otros lenguajes, si el número de parámetros actuales es menor a la cantidad de parámetros formales, algunos de estos quedan con valores nulos.

4.4.1 Correspondencia entre parámetros actuales y formales

Cuando se invoca a un módulo, y este posee parámetros, la correspondencia entre los parámetros actuales y formales puede resolverse, básicamente, de dos formas: **por posición** y **por nombre**.

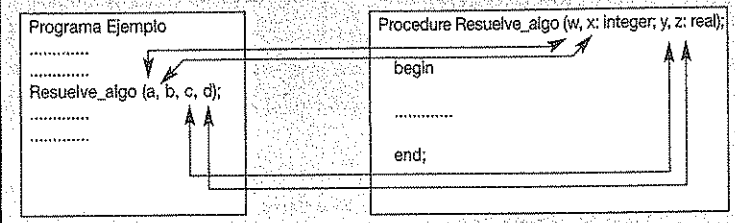
Cuando la correspondencia es por posición, significa que el primer parámetro actual corresponde con el primero formal; el segundo actual con el segundo formal, y así sucesivamente. Esta es la forma más universal de correspondencia y es la adoptada por el lenguaje Pascal.

Ejemplo 4.2

El siguiente segmento de código presenta un módulo con su invocación, la correspondencia de parámetros se muestra en la Figura 4.3.

```
Program ejemplo;
  var a, b: integer;
      c, d: real;
  Procedure resuelve_algo ( w,x: integer; y,z: real );
  begin
    ....
```

```
  end;
begin    {del programa ejemplo }
  ...
  resuelve_algo( a,b,c,d ); ...
end.
```



Nótese que los parámetros actuales a b, que se corresponden con los parámetros formales w x deben tener el mismo tipo de dato, otro tanto ocurre entre c d y y z, enteros y reales respectivamente. Esta coincidencia de tipos es necesaria en el pasaje de parámetros por posición, es decir, el parámetro actual y el parámetro formal deben ser de tipos compatibles.

El pasaje de parámetros por nombre permite indicar en la invocación del módulo con qué parámetro formal se corresponde cada parámetro actual. Esta variante está presente en lenguajes como Ada, si bien también permite la correspondencia por posición. No es objetivo de este libro profundizar sobre el tema, si el lector está interesado en hacerlo puede remitirse a (Sebesta,1996) y (Ghezzi,1999).

4.4.2 Métodos para el pasaje de parámetros

Los métodos para el pasaje de parámetros son formas mediante las cuales los parámetros actuales y formales son transmitidos o devueltos entre los módulos.

Existen varias alternativas para el pasaje de parámetros. En este libro se presentan la variante utilizada por Pascal y por Visual Da Vinci.

Los métodos para pasaje de parámetros que se discutirán son: pasaje por valor, pasaje por resultado, pasaje por valor-resultado y pasaje por referencia. El lector puede profundizar sobre este tema en la bibliografía.

4.4.2.1 Pasaje de parámetros por valor

Cuando un parámetro es pasado por valor, el valor del parámetro actual es (en el módulo que llama) utilizado para inicializar el parámetro formal correspondiente (en el módulo llamado), que

luego actúa como una constante local en el subprograma. Este parámetro se denomina de entrada. La Figura 4.4 a) presenta un gráfico de la semántica de este pasaje.

4.4.2.2 Pasaje de parámetros por resultado

Cuando los parámetros se pasan por resultado, no se transmite ningún valor en el llamado del módulo. El parámetro formal, cuando el módulo finaliza su ejecución, copia su valor al parámetro actual, es decir asigna un resultado para la variable del módulo que llama. Este parámetro se denomina de salida. La Figura 4.4 b) muestra la representación de la semántica del pasaje.

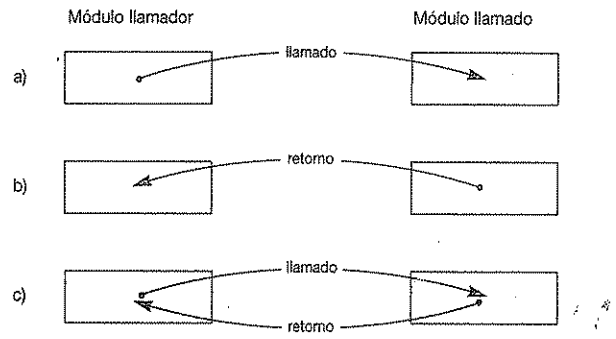


Figura 4.4

4.4.2.3 Pasaje de parámetros por valor resultado

El pasaje de parámetro por valor resultado se denomina de entrada-salida. Es una combinación de los dos métodos anteriores, el valor del parámetro actual es utilizado para inicializar el parámetro formal correspondiente, el cual puede utilizarse como variable local del módulo y, posteriormente, su valor es retornado al parámetro actual (Figura 4.4 c)).

4.4.2.4 Pasaje de parámetros por referencia

Es una segunda implementación del pasaje de entrada-salida. En lugar de transmitir el valor del dato, el parámetro actual transfiere la dirección real de memoria donde la variable se encuentra. De esta forma, el parámetro actual y formal "comparten" la misma zona de memoria. Por consiguiente, cualquier modificación que se realiza en el proceso es "vista" por el modulo llamador.

4.4.3 Implementación de pasaje de parámetros de Pascal

El lenguaje Pascal utiliza como método de pasaje de parámetros una implementación particular del pasaje por valor (para los parámetros de entrada), y el pasaje por referencia (para los parámetros de entrada-salida).

Cuando se transmite una variable a un procedimiento como parámetro por referencia, los cambios que se efectúan en dicha variable, dentro del procedimiento, se mantienen incluso después que este haya terminado. Es decir, los cambios "afectan" (modifica el valor de la variable) al módulo que lo invocó. La Figura 4.5 ilustra este pasaje.

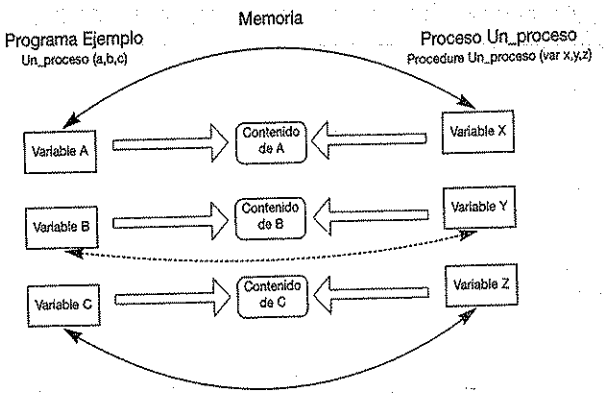


Figura 4.5

La sintaxis que utiliza Pascal para identificar un parámetro por referencia consiste en poner previo a este la palabra clave var.

Código

```
Program ejemplo;
  var a: integer;
      b,c: real;
  Procedure Un_Proceso( var x: integer; var y,z: real );
    var...
  begin
    ....
  end;
begin
  ....
  Un_Proceso( a, b, c);
  ....
end.
```

Nótese que en el ejemplo se pasan 3 parámetros, el primero corresponde al tipo de dato entero y los dos siguientes al tipo de dato real. En este caso, los tres parámetros se pasan por referencia.

La diferencia entre pasaje por valor y por referencia radica en que en el primero guarda en la memoria una copia de la variable, la cual es utilizada por el parámetro formal del módulo. La implementación particular de Pascal no trata al parámetro formal como una constante, sino que permite utilizar al parámetro como una variable más, pero cualquier modificación efectuada no se refleja en el parámetro actual. La Figura 4.6 ilustra este caso.

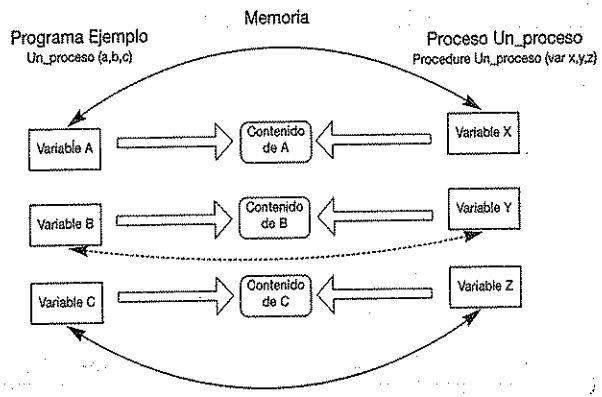


Figura 4.6

La sintaxis de Pascal para los parámetros por valor no utiliza ninguna expresión particular, de aquí que un parámetro por valor se distingue de uno por referencia porque este último tiene la palabra clave var.

Código

```
Program ejemplo;
  var d, e: integer;
      f: real;
  Procedure Un_Proceso( u,v: integer; w: real );
    var...
  begin
    ....
  end;
begin
  ....
  Un_Proceso( d, e, f );
  ....
end.
```

Para ilustrar un procedimiento que utilice pasaje de parámetros por valor y referencia simultáneamente, se define el siguiente programa. Los parámetros actuales g, j se pasan por valor y el parámetro h se pasa por referencia. La Figura 4.7 ilustra el ejemplo.

Código

```
Program ejemplo;
  var g, h: integer;
      j: real;
  Procedure Un_Proceso( g: integer; var h: integer; j: real );
    var...
  begin
    ....
  end;
begin
  ....
  Un_Proceso( g, h, j);
  ....
end.
```

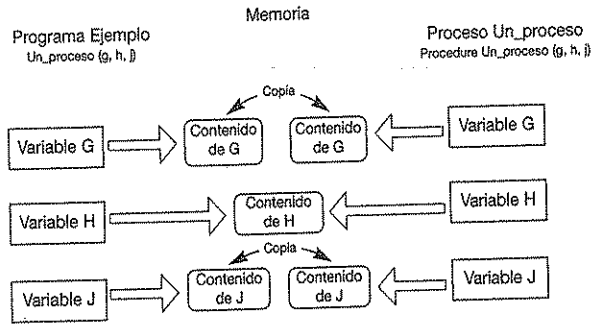


Figura 4.7

Nótese que en este último programa, los identificadores (nombres) de los parámetros formales y actuales coinciden, esto se puede hacer sin problemas en la mayoría de los lenguajes y, en todos los casos, se trata de variables diferentes, ocupan distinto lugar en la memoria y su alcance es distinto.

4.4.3.1 Parámetros en Funciones

La definición de función indicaba que es un módulo que retorna un solo valor asociado al nombre de la misma. Esto indica que una función no debe recibir parámetros por referencia, porque en esos casos se viola la definición conceptual anterior. El programa propuesto presenta la función factorial donde se ilustra la utilización de la misma.

Código

```
Program ejemplo;
  var valor: integer;

  Function Factorial ( dato: integer ): integer;
    var i, calculo: integer;
  begin
    calculo := 1;

    for i := 1 to dato do
      calculo := calculo * i;

    Factorial := calculo;   {se asocia el resultado a la funcion }
  end;

begin
  readln( valor );
  writeln( Factorial( valor ) );
end.
```

4.4.4 Implementación de pasaje de parámetros de Visual Da Vinci

Como se detalló en el capítulo 2, Visual Da Vinci acepta pasaje de parámetros por valor, resultado y valor resultado, implementados como se indicó anteriormente. Las palabras claves para identificar cada uno de estos pasajes es: E (entrada, por valor), S (salida, por resultado) y ES (entrada-salida, por valor resultado). Para más detalles el lector puede consultar el apéndice de Visual Da Vinci o la página web: http://lidi.info.unlp.edu.ar, donde está disponible el ambiente para su utilización.

4.5 Variables locales y variables globales

Como se mencionó anteriormente, una de las características que poseen las variables es el **alcance** o ámbito de referencia de las mismas. El alcance determina la zona del programa donde la variable es definida y conocida. De acuerdo a este alcance, las variables utilizadas en los programa y/o módulos se clasifican en: **locales** y **globales**.

Definición

Una variable local es aquella que está declarada y definida dentro de un programa o módulo, en el sentido que está dentro de ese módulo y es distinta a cualquier variable que tenga el mismo nombre y que estuviera declarada en otro lugar del programa.
Una variable global es aquella que está declarada en el programa y que puede ser utilizada por cualquier módulo de este.

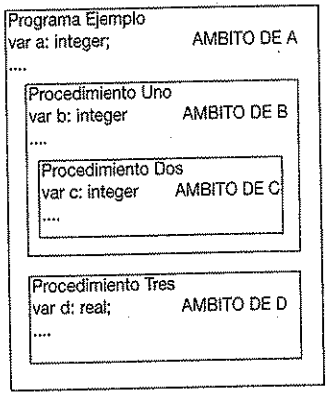


Figura 4.8

El uso de variables locales presenta la ventaja importante de construir módulos altamente independientes, donde las comunicaciones necesarias desde los programas que los invoquen deben hacerse imprescindiblemente mediante el pasaje de parámetros. De esta manera, se evitan posibles efectos laterales producidos por operaciones sobre los datos dentro de un módulo. Es por ello que en el resto de este libro se limitará el uso de variables globales. Cualquier información requerida por un subprograma será pasada como parámetro.

Conceptualmente, puede decirse que desarrollar módulos con independencia funcional, variables locales y mínima comunicación externa mediante parámetros es una buena práctica de programación, que favorece la reutilización y el mantenimiento del software.

4.6 Procedimientos y funciones con parámetros

En esta sección se presentan una serie de ejemplos donde se ilustra la utilización de procedimientos y funciones con pasaje de parámetros.

El primer ejemplo presenta un programa que lee 10 coordenadas (x, y) e informa si cada una pertenece a la recta de ecuación y = 3 x + 2. Para chequear la pertenencia se debe utilizar un módulo. A continuación se presenta una solución al problema planteado.

Código

```
Program coordenadas_x_y;
  var i, x, y: integer;

  Function pertenece_a_la_recta ( x, y: integer ): boolean;
  begin
    pertenece_a_la_recta := ( y = 3 * x + 2 );
    { la instruccion anterior asigna a la funcion un valor booleano
      el mismo resulta de comparar el parametro y con el resultado
      de la cuenta 3 * x + 2 }
  end;

begin
  for i := 1 to 10 do
    begin
      readln( x, y  );
      writeln( pertenece_a_la_recta( x, y ) );
    end;
end.
```

Nótese que se utilizó para la implementación del módulo una función, debido a que el problema de chequear la pertenencia o no del punto en la recta necesita retornar un solo valor como resultado.

El segundo ejemplo pretende realizar un módulo que intercambie contenido de dos variables reales. El siguiente programa muestra una solución de dicho módulo.

Código

```
Procedure Swap( var uno, dos: real );
  var temp: real;   {variable para contener temporariamente un valor}

  begin
    temp  := uno;  {se guarda el valor de la variable uno}
    uno   := dos;  {se asigna el contenido de la variable dos a la uno}
    dos   := temp; {la variable dos recibe el contenido inicial de uno}
  end;
```

Es interesante, para verificar la comprensión de la utilización de parámetros, analizar un algoritmo ya escrito y determinar, en función de las instrucciones que lo componen, qué resultado da en cada caso. Se sugiere al lector analizar el algoritmo del siguiente programa y determinar qué se informa antes de continuar con la lectura del libro.

Código

```
Program var_globales;
  var numero, dato: integer;

  Procedure acumular( a: integer; var b: integer );
    begin
  {a}
      b := 0;
      while( a > 0 ) do
        begin
          b := b + a;
          a := a - 1;
        end;
  {b}
    end;

  begin
    writeln( 'Ingrese un Numero: ' );
    readln( dato );
  {c}
    acumular( dato, numero );
    writeln( 'Dato= ', dato, '     Sumatoria = ', numero );
  end.
```

Se puede analizar qué sucede si se lee el número 5, por ejemplo. El programa informará `Dato=5, Sumatoria=15`, el lector puede repetir el experimento con varios números, y debe llegar a la conclusión que el programa lee números enteros e informa la sumatoria entre 1 y el número entero leído.

Suponiendo que se agrega la sentencia `numero := 0`, en reemplazo del comentario `{a}`, ¿qué sucede ahora? Luego se puede analizar que pasa si la sentencia reemplaza, al comentario `{b}` y al `{c}` respectivamente.

Si se decide agregar la sentencia `numero := 0` en el comentario `{a}` el programa no sufre ningún cambio en su ejecución. El lector debe notar que la variable número del programa y la variable b del proceso comparten la misma zona de memoria, por ser un pasaje de parámetro por referencia. Cualquier cambio que se produzca afectará a la otra.

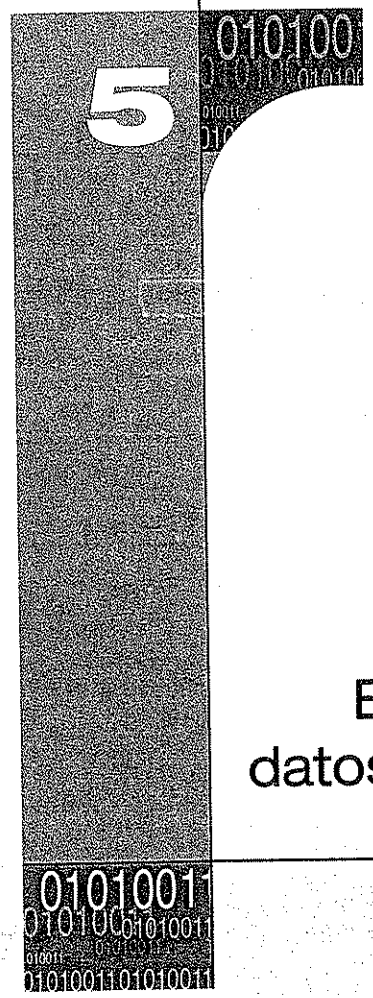
Por el motivo anteriormente expuesto, si la sentencia `numero := 0` se agrega en el comentario `{b}`, el programa informará siempre 0.

Por último, la sentencia agregada reemplazando el comentario `{c}` no modificará el resultado final. El lector debe notar que esta sentencia tiene el mismo efecto que la primera sentencia del módulo `(b:=0)`, por lo tanto, una de las dos estaría de más.

Conclusiones

Se han presentado los conceptos básicos de procedimientos y funciones (en particular en Pascal), resaltando su relación con los conceptos de modularización y reutilización del software.

La ejemplificación apunta a reforzar en el lector el manejo de la comunicación entre módulos vía, pasaje de parámetros, haciendo hincapié en la importancia de utilizar variables locales y minimizar el acoplamiento entre módulos que se da a través de datos compartidos.



Capítulo 5

Estructuras de datos compuestas

5

Estructuras de datos compuestas

Objetivo

En este capítulo se presenta la noción de **tipo de dato compuesto**, el cual permite al usuario definir variables estructuradas que contienen varios valores lógicamente relacionados y asociados bajo un nombre único.

Asimismo, se introduce el concepto de **estructura de datos** y se discuten algunas estructuras de datos básicas (registros, pilas, colas), así como las posibles operaciones con ellas. Se presentan ejemplos de su especificación y su utilización en algoritmos simples.

Por último, el eje temático se encuentra en la definición del concepto de **tipo definido por el usuario**. Esto se aplica a extensiones de pilas y colas, tratando de introducir al lector en la noción básica de **abstracción de datos**.

5.1 Introducción

La principal limitación o "debilidad" que se presenta en los tipos de datos simples (tanto los predefinidos como los definidos por el usuario) es que permiten la creación de variables que representan valores de datos únicos. Por ejemplo, si bien una variable de tipo entero a lo largo de la ejecución de un programa puede tomar diferentes valores, en un momento determinado solo puede contener un número entero.

Todos los tipos estructurados son "moldes" para variables estructuradas, las cuales pueden contener más de un valor. Los tipos de datos estructurados pueden ser construidos a partir de tipos simples o de otros tipos estructurados. De esta forma, la variedad de tipos que pueden definirse es enorme.

Definición

Una estructura de datos es un conjunto de variables (no necesariamente del mismo tipo) relacionadas entre sí de diversas formas.

El usuario es el encargado de definir los tipos estructurados necesarios para resolver cada problema. Estos permiten trabajar con estructuras de datos compuestas para representar los elementos del mundo real, que generalmente son más complejos que un simple número o una letra. Por ejemplo, si es necesario manejar los datos de empleados de una empresa, estos tendrán su nombre, número de documento, fecha de nacimiento, número de legajo, categoría, etc.

En este capítulo se describen y ejemplifican algunas de las estructuras de datos compuestas más conocidas (registros, pilas, colas) que se pueden representar en muchos de los lenguajes actuales. En capítulos posteriores se tratarán otras (como los arreglos de una o más dimensiones y las listas encadenadas). En algunos casos, las estructuras coinciden con tipos compuestos disponibles en el lenguaje (por ejemplo, en los registros y arreglos), mientras que en otros no (por ejemplo, en las listas encadenadas).

Las estructuras de datos compuestas pueden clasificarse teniendo en cuenta diferentes características. Una posibilidad es dividirlas de acuerdo al tipo de los datos que las forman. De esta manera, se pueden distinguir entre estructuras homogéneas y heterogéneas:

Definición

Una estructura de datos es homogénea si los datos que la componen son todos del mismo tipo.

Definición

Una estructura de datos es heterogénea si los datos que la componen son de distinto tipo.

Otra característica que debe considerarse está relacionada con la cantidad de espacio de memoria utilizado por la estructura durante la ejecución del programa. Con este criterio se pueden dividir en estáticas y dinámicas:

Definición

Una estructura de datos es estática si la cantidad de elementos que contienen es fija, es decir, si la cantidad de memoria que se utiliza no varía durante la ejecución de un programa.

Definición

Una estructura de datos es dinámica si el número de componentes y, por lo tanto, la cantidad de memoria, puede variar durante la ejecución de un programa.

Cada una de las estructuras que se describen en este capítulo será clasificada de acuerdo a estas características.

5.2 Registros

Los registros son uno de los tipos de datos estructurados más utilizados. Permiten agrupar datos de diferentes clases y con una conexión lógica en una estructura única. Por ejemplo, los datos correspondientes a un alumno pueden incluir su nombre, número de alumno, documento de identidad, información referente a las materias aprobadas con sus notas, el promedio, etc. Esto puede involucrar datos de tipo entero (por ejemplo, para el número de alumno), de tipo real (por ejemplo, para el promedio), de tipo *string* (por ejemplo, para el nombre), etc. Aunque es posible mantener los datos en forma separada, es más natural y deseable agruparlos de manera que la información referente a un alumno esté contenida en una estructura única. Mientras que manualmente este agrupamiento se puede realizar manteniendo una "ficha" para cada alumno, en un programa esto es factible mediante el uso de registros.

Definición

Un registro es un conjunto de valores, con tres características básicas:

1- Los valores pueden ser de distinto tipo; esto convierte a un registro en una estructura heterogénea.

2- Los valores almacenados en un registro son llamados campos, y cada uno de ellos tiene un identificador; los campos son nombrados individualmente, como variables ordinarias.

3- El almacenamiento ocupado por un registro es fijo; por esto, un registro es una estructura estática.

5.2.1 Declaración de registros

Los tipos registro se construyen identificando al tipo como un registro y luego especificando el nombre y tipo de los campos individuales. Esta lista de campos sigue las reglas generales de declaración de variables, y se encuentra entre las palabras claves `record` y `end`. Los campos pueden ser de cualquier tipo, predefinidos o definidos por el usuario, simples o estructurados.

La forma general de declaración de un tipo registro es:

Código

```
type nombre_tipo_registro = record
     campo_1 : tipo_campo_1;
     campo_2 : tipo_campo_2;
     ...
     campo_n : tipo_campo_n;
     end;
```

donde `nombre_tipo_registro` es el identificador que se le da al tipo, `campo_1, ..., campo_n` son los nombres de los campos del registro, y `tipo_campo_1, ..., tipo_campo_n` son los tipos correspondientes de cada campo.

Como ya se ha expresado en capítulos anteriores, una vez definido el tipo (en este caso un registro) se pueden declarar las variables de ese tipo.

```
var nombre_variable_registro : nombre_tipo_registro
```

Una forma alternativa de lo anterior es realizar la declaración de la estructura del registro directamente con la variable:

Código

```
var nombre_variable_registro = record
     campo_1 : tipo_campo_1;
     campo_2 : tipo_campo_2;
     ...
     campo_n : tipo_campo_n;
     end;
```

De todas maneras, siempre es recomendable definir los tipos y luego declarar variables de esos tipos; esto asegura, entre otras cosas, la legibilidad y la compatibilidad en las asignaciones.

Ejemplo 5.1

Los siguientes son ejemplos de cómo se puede representar distinta información utilizando tipos registro.

```
type racional = record      {tipo de dato para un número racional}
       numerador,
       denominador : integer;
     end;
var numero : racional;      {declaración de la variable}
```

```
type dia_valido = 1 .. 31;
     mes_valido = 1 .. 12;
     anio_valido = 1900 .. 2100;

     fecha = record      {tipo de dato para una fecha}
       dia  : dia_valido;
       mes  : mes_valido;
       anio : anio_valido;
     end;
var dia_de_hoy : fecha;
```

```
type str50 = string[50];
     str30 = string[30];
     libro = record      {tipo de dato para un libro}
       titulo : str50;
       autor : str30;
       anio_edicion : integer;
       editorial : str30;
       ...                 {otros datos del libro}
     end;
var datos_libro : libro;
```

```
type str40 = string[40];
     str10 = string[10];
     cat_valida = 1 .. 15;
     empleado = record      {tipo de dato para un empleado}
       nombre : str40;
       dni : str10;
       categoria : cat_valida;
       ...                 {otros datos del empleado}
     end;
var trabajador : empleado;
```

5.2.2 Acceso a los campos de un registro

Cuando se trabaja con variables simples, la forma de acceder a ellas, ya sea para compararlas o asignarles algún valor, es sencillamente a través de su nombre. Por ejemplo, si a es una variable entera, se le puede asignar valor 10 con una sentencia del tipo `a := 10`.

Esta notación no es suficiente cuando se quiere acceder a un campo de un registro. En este caso, se necesita especificar tanto el nombre del registro como el del campo al que se desea hacer referencia. Esto es lo que se denomina **calificar** el campo. En particular, en Pascal el acceso se realiza de la forma:

```
nombre_variable_registro.nombre_campo
```

donde `nombre_variable_registro` es el nombre de una variable de un tipo registro definido previamente, y `nombre_campo` es el campo del registro que se quiere acceder.

Ejemplo 5.2

Suponiendo las declaraciones de los ejemplos de la sección 5.2.1, las siguientes son acciones válidas:

```
numero.numerador := 18;              { asigna 18 al campo numerador del
                                       registro número }
write (numero.numerador);            { imprime el valor del campo
                                       numerador del registro número }
read (dia_de_hoy.mes);               { lee el valor del campo mes del
                                       registro día_de_hoy }
datos_libro.titulo := 'La Odisea';   { asigna un valor al campo título
                                       del registro datos_libro }
if trabajador.categoria = 6 ...      { evalúa si el valor del campo
                                       categoría del registro trabajador
                                       es 6 }
```

5.2.3 Anidamiento de registros

Las variables estructuradas, como los registros, pueden estar anidadas (una dentro de otra). En otras palabras, un campo de un registro puede, a su vez, ser otro registro.

Ejemplo 5.3

```
type dia_valido = 1 .. 31;
     mes_valido = 1 .. 12;
     anio_valido = 1900 .. 2100;
     str40 = string[40];
     str10 = string[10];
     cat_valida = 1 .. 15;

     fecha = record                  { tipo de dato para una fecha }
       dia  : dia_valido;
       mes  : mes_valido;
       anio : anio_valido;
     end;

     empleado = record               { tipo de dato para un empleado }
       nombre : str40;
       dni : str10;
       categoria : cat_valida;
       fecha_nacimiento : fecha;     { fecha es un registro }
       ...                           { otros datos del empleado }
     end;

var trabajador : empleado;
```

La manera de acceder esencialmente no cambia. En el caso del ejemplo anterior, para acceder a alguno de los campos de `fecha_nacimiento` (día, mes o año), se debe utilizar una doble calificación:

```
trabajador.fecha_nacimiento.dia
```

5.2.4 Operaciones sobre registros

Dado que los campos de un registro son variables de algún tipo de dato, las operaciones posibles sobre un campo son las permitidas para el tipo de dato correspondiente.

Además de las operaciones sobre cada campo, existe una que puede realizarse sobre todo un registro, que es la asignación. Esto es posible si las variables utilizadas en la operación son del mismo tipo registro.

Ejemplo 5.4

Suponiendo la definición del tipo libro dada en la sección 5.2.1, sean las siguientes declaraciones de variables:

```
var datos_libro_1,
    datos_libro_2 : libro;
```

Luego, la asignación:

```
datos_libro_1 := datos_libro_2;
```

es válida.

Debe notarse que, si además de la definición del tipo libro, existiera otra de un tipo `otro_libro` y declaraciones de variables de la forma:

```
type otro_libro = record
          titulo : str50;
          autor : str30;
          anio_edicion : integer;
          editorial : str30;
          ...                        {otros datos }
      end;

var datos_libro_1 : libro;
    datos_otro_libro : otro_libro;
```

entonces, la siguiente asignación no es válida:

```
datos_libro_1 := datos_otro_libro;
```

No pueden realizarse comparaciones entre registros completos. Esto es: dos variables de un mismo tipo registro no pueden ser comparadas con ninguno de los operadores relacionales. Para saber si dos registros son, por ejemplo, iguales, se debe evaluar la igualdad de cada uno de los campos.

Ejemplo 5.5

```
function iguales ( fecha_1, fecha_2 : fecha ) : boolean;
  begin
    iguales :=   (fecha_1.dia = fecha_2.dia) and
                 (fecha_1.mes = fecha_2.mes) and
                 (fecha_1.anio = fecha_2.anio);
  end;
```


Sobre las variables de tipo de dato registro no se pueden aplicar las operaciones de lectura desde teclado y de escritura hacia pantalla. Esto es: no se puede hacer `read(nombre_variable_registro)` ni `write(nombre_variable_registro)`. Obviamente, sí se pueden leer o escribir cada uno de los campos que corresponden a tipos que soportan operaciones de entrada/salida.

Ejemplo 5.6

Sea la definición del tipo empleado de la sección 5.2.3, y la declaración de variable

```
var trabajador : empleado;
```

Luego, las siguientes instrucciones no son válidas:

```
read (trabajador);
write (trabajador);
```

Para leer un registro, puede utilizarse un procedimiento de la forma:

```
procedure leoreg ( var emp : empleado );
  begin
     readln (emp.nombre);
     readln (emp.dni);
     readln (emp.categoria);
     readln (emp.fecha_nacimiento.dia);
     readln (emp.fecha_nacimiento.mes);
     readln (emp.fecha_nacimiento.anio);
     ...
  end;
```

Nótese que las últimas tres lecturas, que permiten obtener la fecha de nacimiento del empleado, no se realizan directamente sobre el campo `fecha_nacimiento` (dado que es un registro) sino sobre cada uno de sus campos.

5.2.5 La sentencia `with`

Cuando se trabaja con registros, hay ocasiones en que el acceso a los campos a través de la calificación suele ser tediosa (por ejemplo, una serie de asignaciones a varios campos, con un largo nombre de registro). Para evitar esto, el lenguaje Pascal provee una sentencia (`with`) que permite que un registro sea nombrado una vez, y luego sea accedido directamente. Su forma general es la siguiente:

Código

```
with nombre_variable_registro do
  begin
    ...     { sentencias que pueden utilizar los campos sin
                    nombrar al registro }
  end;
```

Ejemplo 5.7

Se puede reescribir el procedimiento para leer un registro de la sección anterior utilizando la sentencia with:

```
procedure leoreg_nue ( var emp : empleado );
  begin
    with emp do
      begin
        readln (nombre);
        readln (dni);
        readln (categoria);
        readln (fecha_nacimiento.dia);
        readln (fecha_nacimiento.mes);
        readln (fecha_nacimiento.anio);
        ...
      end
  end;
```

Ejemplo 5.8

Sea el siguiente problema. Se leen datos de notas obtenidas por alumnos de una cátedra, y se desea informar:

- El promedio de notas obtenidas por los alumnos.
- El nombre, el tipo y número de documento y el número de legajo de los alumnos que hayan aprobado con calificación mayor que 7.
- Todos los datos del alumno con peores notas.

La información finaliza cuando se ingresa el alumno 'Roberto Saso', el cual debe ser procesado.

Precondiciones

- Los datos que se ingresan de alumnos son válidos.
- Al menos hay un alumno ingresado.

Poscondiciones

- Alu_peor tiene los datos del peor alumno en notas.
- Si hay dos o más alumnos con igual peor nota, informa el último de ellos.

```
program notas;
  type str40 = string[40];
       str10 = string[10];
```



```
      nota_valida = 0 .. 10;
      alumno = record
        nombre : str40;
        dni : str10;
        legajo : integer;
        nota : nota_valida;
      end;
var alu,
    alu_peor : alumno;
    total_nota,          { para acumular las notas }
    total_alum : integer; { para sumar la cantidad de alumnos }

procedure leoreg( var un_alumno : alumno );
  begin
    with un_alumno do
      begin
        readln (nombre);
        readln (dni);
        readln (legajo);
        readln (nota);
      end;
  end;

  begin
    total_nota := 0;
    total_alum := 0;
    alu_peor.nota := 10;      { nadie puede tener nota > 10 }
    repeat
      leoreg( alu );  { lee información de un alumno }
      total_nota := total_nota + alu.nota; { acumula la nota del
                                             alumno en el total }
      total_alum:= total_alum + 1; { acumula la cantidad de alumnos }
      if alu.nota > 7
        then writeln( 'El alumno: ', alu.nombre, ' DNI ',alu.dni,
                      ' Legajo ', alu.legajo,' aprobó con mas de 7' );
      if alu.nota <= alu_peor.nota
        then alu_peor := alu;
    until (alu.nombre = 'Roberto Saso');
    writeln ('El peor alumno es: ', alu_peor.nombre, ' DNI ', alu_peor.dni,'
             Legajo ', alu_peor.legajo, ' Nota ', alu_peor.nota );
    writeln ( 'El promedio es: ', total_nota / total_alum);
  end.
```

5.3 Pilas

La estructura de datos **pila** *(stack)* es una de las formas más sencillas de organización de datos compuestos: a partir de una dirección de memoria, los datos se almacenan sucesivamente como si fueran una superposición ordenada de elementos (cartas, platos, libros, camisas, etc.), y en cualquier momento se puede recuperar el objeto que se encuentra al tope de la pila, es decir, el último que fue guardado.

La aplicación de este tipo de estructura de datos es muy amplia, y si bien este libro solo abarca algunos ejemplos de tratamiento de secuencias de datos y operadores estructurados en pilas, en informática son numerosas las aplicaciones donde esta estructura es necesaria.

Definición

Una pila es una colección ordenada de elementos, con tres características:

1- Los elementos son del mismo tipo. Esto convierte a la pila en una estructura homogénea.

2- Los elementos pueden recuperarse en orden inverso al que fueron almacenados. Por esto, la forma de acceso a la estructura se denomina LIFO (*Last In First Out*), lo que significa que el último elemento que entró es el primero disponible para salir.

3- La cantidad de elementos que contiene una pila puede variar a lo largo de la ejecución del programa. Por esta razón es una estructura dinámica.

Gráficamente, se puede representar una estructura de datos pila como se muestra en la Figura 5.1.

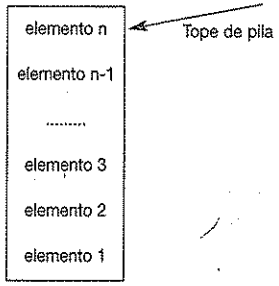


Figura 5.1

Son muy pocos los lenguajes de programación que cuentan con una estructura de este tipo como estandard. En particular, Pascal no dispone del tipo de datos pila.

5.3.1 Declaración de pilas

Como se expresó en la sección anterior, Pascal no dispone entre sus tipos primitivos del tipo de dato pila; por lo tanto, las definiciones que se detallan corresponden a una posible extensión del lenguaje.

La forma general de declaración de un tipo de dato pila es:

```
type nombre_tipo_pila = tipo_de_pila;
```

donde `nombre_tipo_pila` es el identificador que se le da al tipo, y `tipo_de_pila` es alguno de los siguientes valores:

- `stack_integer`
- `stack_real`
- `stack_char`
- `stack_string (donde se supone que cada string ocupa 255 lugares)`
- `stack_rec_tipo_registro (donde tipo_registro es el identificador del tipo de un registro definido previamente).`

Como ya se expresó, una vez definido un tipo pila, pueden declararse variables de ese tipo. Una manera alternativa es realizar la declaración directamente con la variable:

```
var nombre_variable_pila : tipo_de_pila;
```

De todas maneras, siempre es recomendable definir los tipos y luego las variables de esos tipos, ya que esto facilita, como ya se indicó en casos anteriores, la legibilidad y la compatibilidad en las asignaciones.

Ejemplo 5.9

```
type  pila_de_enteros = stack_integer;
      pila_de_reales = stack_real;
      registro = record
        ...
      end;
      pila_de_registros = stack_rec_registro;
var   pila1 : pila_de_enteros;      {declara una pila de enteros}
      pila2 : pila_de_reales;       {declara una pila de reales}
      pila3 : pila_de_registros;    {declara una pila de registros}
```

5.3.2 Operaciones sobre pilas

La primera operación a realizar sobre una pila es su "activación" o creación: esto significa asignarle efectivamente una dirección de inicio en la memoria y que se pueda comenzar a operar sobre ella. Esto se realiza de la forma:

```
st_create( nombre_variable_pila )
```

donde `nombre_variable_pila` es una variable de tipo pila. Esta sentencia realiza la creación de una pila de nombre `nombre_variable_pila`.

Ejemplo 5.10

De acuerdo a las definiciones anteriores, son válidas:

```
st_create ( pila1 );
st_create ( pila2 );
st_create ( pila3 );
```

También son necesarias: una operación para agregar elementos a la pila (apilar) y otra para extraerlos (des-apilar).

La operación de apilado tiene la siguiente sintaxis:

```
st_push( nombre_variable_pila, elemento )
```

donde `nombre_variable_pila` es el nombre de una variable de tipo pila, y elemento es una variable del mismo tipo que el declarado para los elementos de la pila.

La semántica de esta operación es la transferencia del contenido de elemento al tope de la pila, incrementando en uno el tamaño actual de la misma.

Ejemplo 5.11

Sea el siguiente problema: almacenar en una pila los 10 primeros números naturales, y en otra pila sus correspondientes raíces cuadradas. Una posible solución es el siguiente programa:

```
program pila_uno;
  type pila_entera = stack_integer;
       pila_real = stack_real;

  var pe : pila_entera;
      pr : pila_real;
      j  : integer;
      r  : real;

  begin
    st_create( pe );
    st_create( pr );
    for j := 1 to 10 do
      begin
        st_push( pe, j );       {apila el número j}
        r := sqrt( j );
        st_push( pr, r );       {apila la raíz cuadrada de j}
      end;
  end.
```


La operación de des-apilado tiene la siguiente sintaxis:

```
st_pop( nombre_variable_pila, elemento )
```

donde `nombre_variable_pila` es el nombre de una variable de tipo pila, y elemento es una variable del mismo tipo que el declarado para los elementos de la pila.

El efecto de esta operación es el de transferir el último elemento ingresado a la pila (el que está en el tope) a la variable elemento, decrementando en uno el tamaño actual de la pila. Nótese que para que esta operación sea válida, la pila debe contener al menos un elemento.

Ejemplo 5.12

Leer una sucesión de 12 caracteres e imprimirlos en orden inverso al leído.

Precondiciones

- **La pila inicial está vacía.**
- **Se ingresan exactamente doce caracteres.**

Poscondiciones

- **La pila final se encuentra vacía.**

```
program pila_dos;
  type pila_caracter = stack_char;

  var pc : pila_caracter;
      c  : char;
      j  : integer;

  begin
    st_create( pc );
    { lee los 12 caracteres y los almacena en una pila }
    for j := 1 to 12 do
      begin
        read ( c );
        st_push ( pc, c );
      end;
      { recupera los caracteres desde la pila y los imprime }
    for j := 1 to 12 do
      begin
        st_pop ( pc, c );
        write(c);
      end;

  end.
```

La operación de des-apilado (st_pop) requiere que la pila no esté vacía. En el ejemplo anterior, se asumió conocida (como precondición) la cantidad de veces que se debe desapilar, pero esta no es la situación en la mayoría de los casos. Por lo tanto, es necesario contar con una operación que permita saber si la pila está vacía. Esto se realiza mediante la función st_empty, la cual retorna un valor lógico true si la pila no contiene elementos, y falso en caso contrario:

```
resultado := st_empty( nombre_variable_pila )
```

donde nombre_variable_pila es el nombre de una variable de tipo pila, y resultado es una variable de tipo boolean.

Ejemplo 5.13

Leer una sucesión de nombres de personas, que finaliza con 'Roberto Saso' (el cual debe procesarse), e imprimirlos en orden inverso al leído.

Precondiciones:

- La pila inicial se encuentra vacía.
- Al menos se ingresa el dato 'Roberto Saso'

Poscondiciones:

- La pila final se encuentra vacía.
- El primer dato que debe imprimir es 'Roberto Saso'

```
program pila_tres;
  type pila_string = stack_string;

  var  ps : pila_string;
       s : string;

  begin
    st_create( ps );
    { lee los nombres y los almacena en una pila }
    repeat
        readln ( s );
        st_push ( ps, s );
    until (s = 'Roberto Saso');
    { recupera los nombres desde la pila y los imprime }
    while not ( st_empty ( ps ) ) do
      begin
        st_pop ( ps, s );
        writeln(s);
      end;

  end.
```


En la resolución de algunos problemas, puede ser interesante conocer la cantidad de elementos que se encuentran almacenados en una pila. Esto puede realizarse mediante la operación st_length, de la forma:

```
cantidad := st_length( nombre_variable_pila )
```

donde nombre_variable_pila es el nombre de una variable de tipo pila, y cantidad es una variable entera.

Ejemplo 5.14

Sea el problema del ejemplo anterior. Una forma alternativa de resolución utilizando la operación st_length es la siguiente:

```
program pila_cuatro;
  type pila_string = stack_string;

  var  ps : pila_string;
       s : string;
       j,
       cant : integer;

  begin
    st_create( ps );
    { lee los nombres y los almacena en una pila }
    repeat
        readln ( s );
        st_push ( ps, s );
    until (s = 'Roberto Saso');
    { recupera los nombres desde la pila y los imprime }
    cant := st_length ( ps );
    for j := 1 to cant do
      begin
        st_pop ( ps, s );
        writeln(s);
      end;

  end.
```

Por último, existe una operación que permite examinar el contenido del elemento que se encuentra al tope de la pila, pero sin extraerlo de la misma. Esta operación es de la forma:

```
elemento = st_top( nombre_variable_pila )
```

donde nombre_variable_pila es el nombre de una variable de tipo pila, y elemento es una variable del mismo tipo que el declarado para los elementos de la pila.

Ejemplo 5.15

Leer un conjunto de valores enteros, terminados en 0, e indicar cuántos de ellos son capicúa. Se leen números enteros, el número cero no debe procesarse.

```
program capicua;
  type pila_enteros = stack_integer;

  var numero,
      cantidad : integer;        { para leer cada número }

  { esta función recibe un número y retorna otro con las cifras invertidas }
  function descomponer ( nro: integer ): integer;
    var prod,
        digito,
        resto: integer; {para obtener cada dígito del número}
        pe : pila_enteros;

    begin
      { descompone el número recibido en cada una de sus cifras y guarda
        en la pila }
      st_create( pe );
      while (nro <> 0 ) do
        begin
          resto := nro mod 10;
          st_push( pe, resto );
          nro := nro div 10;
        end;
      { rearma el número }
      prod := 1;                {la unidad tiene peso 1}
      while not ( st_empty( pe ) ) do
        begin
          st_pop( pe, digito ); {obtiene el primer dígito }
          nro := nro + digito * prod;
          {se multiplica cada dígito por su peso dentro del
           número(unidad, decena,....)}
          prod := prod * 10; {el peso del siguiente dígito será 10
                              veces mayor}
        end;
      descomponer := nro;
    end;

  begin
    readln ( numero );
    cantidad := 0;        {cuenta la cantidad de capicúas}
    while ( numero <> 0 ) do
      begin
```



```
        if numero = descomponer( numero ) then cantidad :=
                    cantidad + 1;
        read( numero );
      end;
    writeln( 'La cantidad de números capicúas leídos es:', cantidad );
  end.
```

Ejemplo 5.16

Leer una sucesión de palabras de hasta 10 caracteres, separadas por blancos (las palabras se leen caracter por caracter). Todo el texto termina en punto (el punto no es un caracter permitido en las palabras). Se desea obtener, en primera instancia, cada palabra leída en forma invertida. Por ejemplo: se lee "casa" y se imprime "asac".

Comentario: una pila sirve para guardar caracter por caracter una palabra. La estructura de pila permite recuperar naturalmente la palabra invertida. Para "armar" la palabra, a medida que se recuperan los caracteres de la pila se forma un *string* de largo máximo 10, donde queda la palabra a imprimir.

```
program invertir_palabras;
  type pila_caracter = stack_char;
       str10 = string[30];

  var pila_caracter;
      pal : str10;
      letra : char;

  begin
    st_create( pc );
    read ( letra );
    while ( letra <> '.' ) do
      begin
        while ( letra <> ' ') do {arma la pila con una palabra}
          begin
            st_push( pc, letra );
            read ( letra );
          end;
        pal := '';        {aquí se va a ir armando la palabra}
        while not ( st_empty( pc ) ) do {arma palabra invertida}
          begin
            st_pop( pc, letra );
            pal := pal + letra;
          end;
        writeln( pal );
        read ( letra );
      end;
  end.
```

A continuación se plantean una serie de inquietudes a resolver:

- ¿Qué sucede si el texto se inicia con 5 blancos?
- ¿Qué sucede si entre la última palabra y el punto no hay blancos?
- ¿Qué sucede si aparece una palabra de más de 10 letras?

5.4 Colas

La estructura de datos **cola** *(queue)* es otra de las formas más sencillas de organización de datos compuestos. A partir de una dirección de memoria, los datos se almacenan consecutivamente como una sucesión de elementos y en cualquier momento se puede recuperar el objeto que se encuentra primero en la cola, es decir, el primero que fue guardado.

La aplicación de este tipo de estructura de datos es muy amplia, y es fundamental entender que muchos fenómenos de la vida real suceden "con colas" (por ejemplo, el tránsito, una línea de producción en una fábrica, una cola de tareas a realizar por un procesador, la atención en un Banco, etc.). Si bien este libro solo trata algunos ejemplos de tratamiento de secuencias de datos y operadores estructurados en colas, son numerosas las aplicaciones en que es necesario procesar la información estructurada en colas.

Definición

Una cola es una colección ordenada de elementos, con tres características:

1- Los elementos son del mismo tipo. Esto convierte a la cola en una estructura homogénea.

2- Los elementos pueden recuperarse en el orden en que fueron almacenados. Por esto, la forma de acceso a la estructura se denomina FIFO (*First In First Out*), lo que significa que el primer elemento que entró es el primero disponible para salir.

3- La cantidad de elementos que contiene una cola puede variar a lo largo de la ejecución del programa. Por esta razón es una estructura dinámica.

Gráficamente, se puede representar una estructura de datos cola como se muestra en la Figura 5.2.

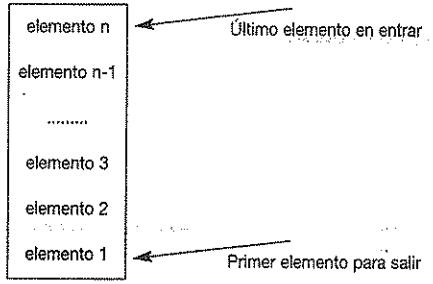


Figura 5.2

Son muy pocos los lenguajes de programación que cuentan con una estructura de este tipo como tipo de dato estándar. En particular, Pascal no dispone del tipo de datos cola.

5.4.1 Declaración de colas

Como se expresó en la sección anterior, Pascal no dispone entre sus tipos primitivos del tipo de dato cola; por lo tanto, las definiciones que se detallan corresponden a una posible extensión del lenguaje.

La forma general de declaración de un tipo de dato cola es:

```
type nombre_tipo_cola = tipo_de_cola;
```

donde `nombre_tipo_cola` es el identificador que se le da al tipo, y `tipo_de_cola` es alguno de los siguientes valores:

- `queue_integer`
- `queue_real`
- `queue_char`
- `queue _string` (donde se supone que cada string ocupa 255 lugares)
- `queue _rec_tipo_registro` (donde `tipo_registro` es el identificador del tipo de un registro definido previamente).

Como ya se expresó, una vez definido un tipo cola, pueden declararse variables de ese tipo. Una manera alternativa de lo antedicho es realizar la declaración directamente con la variable:

```
var nombre_variable_cola : tipo_de_cola;
```

De todos modos, siempre es recomendable definir los tipos y luego variables de esos tipos, ya que esto facilita, como ya se indicó en casos anteriores, la legibilidad y la compatibilidad en las asignaciones.

Ejemplo 5.17

```
type  cola_de_enteros = queue_integer;
      cola_de_reales = queu_real;
      registro = record
          ...
      end;
      cola_de_registros = queue_rec_registro;
var   cola1 : cola_de_enteros;          {declara una cola de enteros}
      cola2 : cola_de_reales;           {declara una cola de reales}
      cola3 : cola_de_registros;{declara una cola de registros}
```

5.4.2 Operaciones sobre colas

La primera operación a realizar sobre una cola es su "activación" o creación: esto significa asignarle efectivamente una dirección de inicio en la memoria y que se pueda comenzar a operar sobre ella. Esto se realiza de la forma:

```
q_create( nombre_variable_cola )
```

donde `nombre_variable_cola` es una variable de tipo cola. Esta sentencia realiza la creación de una cola de nombre `nombre_variable_cola`.

Ejemplo 5.18

Con las definiciones anteriores, son válidas:

```
q_create ( pila1 );
q_create ( pila2 );
q_create ( pila3 );
```

También son necesarias, una operación para agregar elementos a la cola (encolar) y otra para extraerlos (desencolar).

La operación de encolado tiene la siguiente sintaxis:

```
q_push( nombre_variable_cola, elemento )
```

donde `nombre_variable_cola` es el nombre de una variable de tipo cola, y `elemento` es una variable del mismo tipo que el declarado para los elementos de la cola.

El efecto de esta operación es transferir el contenido de `elemento` al final de la cola, incrementando en uno el tamaño actual de la misma.

Ejemplo 5.19

Sea el siguiente problema: almacenar en una cola los 10 primeros números naturales, y en otra cola sus correspondientes raíces cuadradas. El algoritmo es similar al primer ejemplo de pilas; nótense las diferencias y similitudes.

```
program cola_uno;
  type cola_entera = queue_integer;
       cola_real = queue_real;

  var ce : cola_entera;
      cr : cola_real;
      j : integer;
      r : real;

  begin
    q_create( ce );
    q_create( cr );
    for j := 1 to 10 do
      begin
        q_push( ce, j );  { encola el número j }
        r := sqrt( j );
        q_push( cr, r );  { encola la raíz cuadrada del número j }
      end;
  end.
```


La operación de des-encolado tiene la siguiente sintaxis:

```
q_pop( nombre_variable_cola, elemento )
```

donde `nombre_variable_cola` es el nombre de una variable de tipo cola, y `elemento` es una variable del mismo tipo que el declarado para los elementos de la cola.

El efecto de esta operación es transferir el primer elemento ingresado a la cola a la variable `elemento`, decrementando en uno el tamaño actual de la cola. Nótese que para que esta operación sea válida, la cola debe contener al menos un elemento.

Ejemplo 5.20

Leer una sucesión de 15 nombres y notas de alumnos. Informar los nombres y notas de los alumnos que superan el promedio del grupo.

```
program cola_dos;
  type str30 = string[30];
       nota_valida = 0 .. 10;
       datos = record
         nombre : str30;
         nota : nota_valida;
       end;
       cola_datos = queue_rec_datos;

  var cd : cola_datos;
      alu : datos;
      total_nota : integer;    {para acumular las notas}
      j,
      prom : real;
```

```
procedure leoreg ( var un_alumno : datos );
  begin
    with un_alumno do
      begin
        readln (nombre);
        readln (nota);
      end;
  end;

  begin
    q_create ( cd );
    {lee los datos, acumula las notas y los almacena en una cola}
    total_nota := 0;
    for j := 1 to 15 do
      begin
        leoreg( alu );   {lee información de un alumno}
        q_push ( cd, alu );
        total_nota := total_nota + alu.nota;
        {acumula la nota del alumno en el total}
      end;
    prom := total_nota / 15;
    {recupera los datos desde la cola, testea y, si corresponde, imprime}
    for j := 1 to 15 do
      begin
        q_pop ( cd, alu );
        if alu.nota > prom then writeln ('El alumno ',
                  alu.nombre, ' tuvo nota ', alu.nota);
      end;
  end.
```

La operación de des-encolado (`q_pop`) requiere que la cola no esté vacía. En el ejemplo anterior se conocía la cantidad de elementos que iba a contener (15), pero esta no es la situación en todos los casos. Por esto, es necesario contar con una operación que permita saber si la cola está vacía. Esto se realiza mediante la función `q_empty`, la cual retorna un valor lógico `true` si la cola no contiene elementos, y `false` en caso contrario:

```
resultado := q_empty( nombre_variable_cola )
```

donde `nombre_variable_cola` es el nombre de una variable de tipo cola, y `resultado` es una variable de tipo boolean.

Ejemplo 5.21

Sea el problema anterior, pero ahora no se conoce *a priori* la cantidad de datos que se leen; solo se sabe que el último en ser procesado es el alumno 'Roberto Saso'.

```
program cola_tres;
  type ...  {las mismas definiciones que para cola_dos}

  var ...   {las mismas que para cola_dos}

  procedure leoreg ( var un_alumno : datos );
      ...          { idem que para cola_dos }

  begin
    q_create ( cd );
    {lee los datos, acumula las notas y los almacena en una cola}
    total_nota := 0;
    j := 0;
    repeat
      leoreg( alu );    {lee información de un alumno }
      q_push ( cd, alu );
      total_nota := total_nota + alu.nota;
      {acumula la nota del alumno en el total}
      j := j + 1;
    until s = 'Roberto Saso';
    prom := total_nota / j; { recupera los datos desde la cola, testea
                              y, si corresponde, imprime}
    while not (q_empty (cd) ) do
      begin
        q_pop ( cd, alu );
        if alu.nota > prom then writeln ('El alumno ', alu.nombre,
            ' tuvo nota ', alu.nota);
      end;
  end.
```

En la resolución de algunos problemas, puede ser interesante conocer la cantidad de elementos que se encuentran almacenados en una cola. Esto puede realizarse mediante la operación `q_length`, de la forma:

```
cantidad := q_length( nombre_variable_cola )
```

donde `nombre_variable_cola` es el nombre de una variable de tipo cola, y `cantidad` es una variable de tipo entero.

Ejemplo 5.22

Sea el problema del ejemplo anterior. Una forma alternativa de resolución utilizando la operación q_length es la siguiente:

```
program cola_cuatro;
  type ... {las mismas definiciones que para cola_dos}

  var cant : integer;
          ...              {las mismas que para cola_dos}

  procedure leoreg ( var un_alumno : datos );
          ...            {idem que para cola_dos}

  begin
    q_create ( cd );
    {lee los datos, acumula las notas y los almacena en una cola}
    total_nota := 0;
    j := 0;
    repeat
      leoreg( alu );     {lee información de un alumno}
      q_push ( cd, alu );
      total_nota := total_nota + alu.nota;
              {acumula la nota del alumno en el total}
    until s = 'Roberto Saso';
    prom := total_nota / q_length (cd);   {notar este cambio}
    {recupera los datos desde la cola, testea y, si corresponde, imprime}
    cant := q_length (cd);
    for j := 1 to cant do
      begin
        q_pop ( cd, alu );
        if alu.nota > prom then writeln ('El alumno ', alu.nombre,
          ' tuvo nota ', alu.nota);
      end;
  end.
```

Existe una operación que permite examinar el contenido del elemento que se encuentra al principio de la cola, pero sin extraerlo de la misma. Esta operación es de la forma:

```
elemento := q_top( nombre_variable_cola )
```

donde `nombre_variable_cola` es el nombre de una variable de tipo cola, y `elemento` es una variable del mismo tipo que el declarado para los elementos de la cola.

Por último, se tiene una operación que solamente está definida para las colas y que permite examinar el último elemento ingresado, sin extraerlo. Esta operación es de la forma:


```
elemento :=q_bottom( nombre_variable_cola )
```

donde `nombre_variable_cola` es el nombre de una variable de tipo cola, y `elemento` es una variable del mismo tipo que el declarado para los elementos de la cola.

Ejemplo 5.23

Un supermercado cuenta con 3 cajas para el cobro de las compras de los clientes. De cada cliente se conoce la cantidad de artículos comprados. En un determinado momento una de las cajas cierra. Escriba un módulo que dadas las tres colas correspondientes a las cajas, distribuya los clientes que están en la caja que cierra en las dos restantes, agregando los que tienen menos de diez productos en la primera cola y el resto en la segunda.

```
type cola_super = queue_integer;

procedure tres_a_dos ( var cola1, cola2, cola3 : cola_super );
  var articulos : integer;
  begin
    while not ( q_empty (cola3) ) do
      begin
        q_pop ( cola3, articulos );
        if articulos < 10
          then q_push ( cola1, articulos )
          else q_push ( cola2, articulos );
      end;
  end;
```

Ejemplo 5.24

Se quiere representar el fenómeno de atención en una ventanilla de servicios. Delante de esta ventanilla se forma una cola de usuarios caracterizados por su nombre y apellido, dirección, código de impuesto a pagar, monto a pagar, y tiempo de atención previsto (en segundos). Una vez formada la cola (hasta 100 usuarios o hasta que llega el usuario 'zzzz'), se debe simular la atención, informando:

- total de usuarios atendidos y monto recaudado al final de la atención
- tiempo total transcurrido (en minutos) para la atención de todos los usuarios

Comentarios: la estructura de datos requerida es una cola de registros, donde cada elemento es un registro de usuario.

```
program atencion;
  type str40 = string[40];
       usuario = record
         nombre : str40;
         direccion : str40;
```

```
        codigo : integer;
        monto : real;
        t_aten : integer;
      end;
      cola_servicios = queue_rec_usuarios;

var cola : cola_servicios;
    persona : usuario;
    tiempo_tot,
    atendidos : integer;
    monto_tot : real;

procedure Armar_cola ( var co : cola_servicios );
  var us : usuario;
      cant : integer;

  procedure leoreg ( var usu : usuario );
    begin
      with usu do
        begin
          readln (nombre);
          readln (direccion);
          readln (codigo);
          readln (monto);
          readln (t_aten);
        end;
    end;

    begin
      q_create (co);
      cant := 0;
      leoreg (us);
      while ( cant < 100 ) and ( us.nombre <> 'ZZZZ' ) do
        begin
          q_push ( co, us);
          leoreg (us);
        end;
    end;

begin
  tiempo_tot := 0;
  atendidos := 0;
  monto_tot := 0;
  Armar_cola ( cola );
  while not ( q_empty( cola ) ) do
    begin
      q_pop( cola, persona );                {"atiende" a la persona}
      tiempo_tot := tiempo_tot + persona.t_aten; { acumula el tiempo }
      monto_tot := monto_tot + persona.monto;{ acumula el monto }
      atendidos := atendidos + 1;            {atendió uno más}
    end;
```



```
  { Imprime los datos finales }
  writeln( 'Total de usuarios atendidos: ', atendidos );
  writeln( 'Monto total recaudado: ', monto_tot );
  writeln( 'Tiempo total de atención: ', tiempo_tot );
end.
```

Finalmente, se plantean dos inquietudes a resolver:

- ¿Cómo podría atender hasta un tiempo total máximo?
- ¿Cómo se podría informar el "estado" de la cola cada 10 minutos? (es decir, el número de personas que restan atender y su tiempo requerido)

5.5 Concepto de tipo definido por el usuario. Extensiones a pilas y colas.

Al presentar los tipos de datos pila y cola se indicó que pocos lenguajes de programación cuentan con estas estructuras como tipo de dato estándar. En particular, Pascal no las brinda. Por este motivo, para disponer de ellas, es el usuario quien debe definirlas.

Definición

Un tipo definido por el usuario consta de un tipo de datos base y un conjunto de operaciones (biblioteca) que pueden realizarse sobre él.

Con esta definición, las pilas y colas utilizadas en secciones anteriores son tipos definidos por el usuario, ya que presentan un tipo y un conjunto de operaciones sobre el mismo.

De la misma manera, pueden obtenerse otros tipos definidos por el usuario. Esto puede hacerse mediante la utilización de un tipo base distinto y/o modificaciones sobre el conjunto de operaciones.

Esto adquiere particular importancia para modelizar adecuadamente el mundo real ya que es posible especificar los elementos reales de acuerdo a sus características como tipos definidos por el usuario.

Por ejemplo, pueden definirse los tipos cola doble (constituida por un tipo base con dos colas, y donde las operaciones incluyen alguna que permita pasar de una cola a otra), cola bancaria (donde podría tenerse una única cola y varias ventanillas, y la atención se haría en alguna de las ventanillas), etc. Se deja al lector la definición e implementación de estos tipos.

Conclusiones

En este capítulo se presentó el concepto de estructuras de datos compuestas y algunas de ellas, como los registros, pilas y colas. También se ejemplificó el uso de las operaciones sobre estas estructuras en algoritmos sencillos.

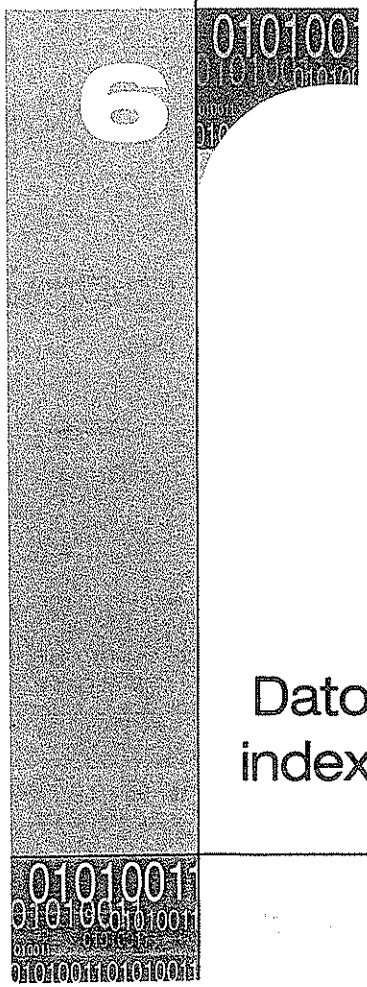
Además, es importante la introducción de la idea de tipo definido por el usuario, por dos aspectos: por un lado, como una manera de representar adecuadamente los elementos del mundo real y, por otra parte, como una noción básica de abstracción de datos. Este tema es tratado más ampliamente en el capítulo referido a tipos abstractos de datos.

Ejercicios

1. En cada caso defina la estructura apropiada para almacenar la información asociada: una carta de un mazo de cartas, una fecha, un horario, una agenda, un artículo del *stock* de un comercio, un alumno de un colegio secundario.
2. Se lee información del pago de cuotas, de la forma Nombre y Apellido, Ciudad, Número de cuota y Monto abonado. La información recibida finaliza al procesar a "Roberto Saso". Se quiere obtener:
   - el monto total recaudado.
   - el nombre y apellido de las personas cuya ciudad es "La Plata".
   - el porcentaje que representan los pagos de la cuota 22 sobre el total.
3. Se desea conocer datos estadísticos referentes al movimiento de ómnibus en una terminal, para lo cual se procesa información de los servicios. De cada uno se conoce: la empresa, el destino, la cantidad total de asientos y el número de pasajeros. Realice un programa para obtener: la empresa de la que partieron la mayor cantidad de ómnibus con pasaje completo, la empresa con menor promedio de pasajeros y la cantidad de servicios que partieron a Bariloche. La información referida a una empresa se lee en forma sucesiva, y el último que se procesa es el ómnibus de la empresa "Rio de La Plata" con destino a "San Bernardo".
4. Escriba un programa que lea una secuencia de caracteres terminada en '.' y verifique si es de la forma @ # @', donde @ es una serie de letras minúsculas, # es el carácter '#', y @' es la inversa de la primera secuencia de letras, pero con las letras en mayúsculas. Ejemplo: la secuencia 'sewdgoh#HOGDWES' es de la forma descrita.
5. Realizar un módulo que reciba una palabra y devuelva la cantidad de veces que la misma aparece invertida en un texto que se lee y termina con punto. Por ejemplo, dada la palabra 'sal' y el texto 'las remeras y las flores se las dio a las dueñas', la cantidad de veces es 4.
6. Dadas dos secuencias de caracteres separadas por '*' determinar si las mismas son iguales.

BIBLIOTECA FAC. DE INFORMÁTICA U.N.L.P.

7. El día de pago a los docentes, un Banco decide habilitar dos cajas. Supóngase que cada persona sabe exactamente cuánto tiempo tarda el cajero en pagarle (este tiempo puede ser distinto para cada persona ya que depende de la cantidad de cheques que ésta tenga). Para facilitar el problema supóngase que todos los docentes llegan a la cola antes de las 10 hs., es decir, antes de que el Banco comience a atender.
   a) Escriba un módulo con el siguiente encabezado:
   ```
   Procedure atención (tiempo : integer, var cola : q_reg);
   ```
   que simule la atención de la cola durante el tiempo pasado como parámetro. Lo que realiza este módulo es des-encolar elementos hasta que la suma de los tiempos de atención supere el tiempo indicado.
   b) Informar el estado de las colas a las 11 hs.
   c) Informar la cantidad de personas que quedan sin atender en cada ventanilla (cuando el banco cierra a las 15 hs.).
   d) Informar cuántas personas fueron atendidas.
   e) ¿Cuáles son los nombres de las personas que quedaron al frente de cada ventanilla al terminar la atención (si existen)?



Capítulo 6

Datos compuestos indexados: arreglos

6

Datos compuestos indexados: arreglos

Objetivo

En este capítulo se presentan los **tipos de datos compuestos indexados**, introduciendo la estructura de datos **arreglo**. A diferencia de las estructuras de datos descritas en el capítulo anterior (pilas y colas), en las cuales solo se tiene acceso a un elemento (el primero o el último ingresado, según el caso), los arreglos permiten operar sobre cualquier elemento de la estructura, especificando su **posición** en la misma.

Se presentan los arreglos de una dimensión (conocidos como vectores), de dos dimensiones (matrices) y, en general, los arreglos n-dimensionales, con numerosos ejemplos de algoritmos de aplicación que utilizan dichas estructuras.

Por último se hace una comparación conceptual de las estructuras compuestas indexadas con las pilas y colas vistas en el capítulo anterior, extendiendo dicha comparación a la resolución de problemas similares utilizando diferentes estructuras de datos.

6.1 Clasificación de las estructuras de datos

Como se definió anteriormente, una estructura de datos es la representación de un objeto o elemento del problema dentro del algoritmo.

Las estructuras de datos pueden ser clasificadas de diversas formas, de acuerdo al tipo de datos que representan. Estas clasificaciones pueden ser:

Estructuras de datos simples o compuestas.

Las estructuras de datos simples representan un único valor, por ejemplo, un número entero o un carácter. Las estructuras de datos compuestas pueden contener más de un valor como, por ejemplo, un registro o una pila.

Estructuras de datos homogéneas o heterogéneas.

Se llama homogéneas a aquellas estructuras de datos compuestas que tienen todos sus elementos del mismos tipo como, por ejemplo, las pilas o las colas. Por otra parte, las heterogéneas son aquellas estructuras de datos compuestas cuyos elementos pueden ser de distinto tipo como, por ejemplo, los registros.

Estructuras de datos estáticas o dinámicas.

Una estructura de datos se denomina estática si la cantidad de elementos que contiene es fija, es decir, si la cantidad de memoria que se utiliza no varía durante la ejecución de un programa. Una estructura de datos se denomina dinámica si el número de componentes y, por lo tanto, la cantidad de memoria puede variar durante la ejecución de un programa. Las estructuras estáticas tienen como desventaja el mal aprovechamiento de la memoria, ya que si contienen pocos elementos se desperdicia lugar, así como tampoco tienen posibilidad de recibir más elementos de los indicados inicialmente. Pero presentan una gran ventaja, que es su reserva inicial de memoria y el acceso directo a cualquiera de sus elementos, ya que la posición de cualquiera de ellos puede calcularse como un desplazamiento de la posición inicial de la estructura. A esta clasificación pertenecen, por ejemplo, los arreglos que se verán en este capítulo. Por su parte, las estructuras dinámicas tienen la ventaja de poder reservar la memoria necesaria para cada uno de sus elementos a medida que el algoritmo lo requiere; esto implica un mejor uso de la memoria y la posibilidad de extender la estructuras según las necesidades del problema (respetando las limitaciones del hardware). Pero como contrapartida, los datos deben ser accedidos en forma secuencial dado que sus posiciones no se encuentran consecutivas físicamente en la memoria. A esta clasificación pertenecen, por ejemplo, las listas y árboles que se verán en el capítulo 10.

Por otra parte, el manejo dinámico de la memoria obliga a un mayor esfuerzo por parte del lenguaje y el sistema operativo en lo referido a la recuperación de los espacios que pasan de ocupados a libres, con el consiguiente incremento de tiempo.

La elección del tipo de estructura de datos necesaria para resolver un problema depende, básicamente, de cual es la más adecuada al problema y a las restricciones del equipamiento y lenguaje de programación disponible.

6.2 Arreglos

Como se definió en el capítulo anterior, tanto las pilas como las colas son estructuras de datos homogéneas, cuyos elementos pueden ser accedidos solo desde sus extremos y que, por lo tanto, tienen un orden interno muy rígido. Además, representan estructuras dinámicas, pues la memoria ocupada depende de las sucesivas operaciones de incorporación y eliminación de elementos que se realizan sobre ellas.

En una variedad de aplicaciones resulta de gran interés poder acceder libremente a cada uno de los elementos que componen una estructura de datos, sin verse obligado a tener que acceder a los restantes. La estructura de datos **arreglo** ***(array)*** permite esta forma de acceso.

Definición

Un tipo de dato arreglo es una colección ordenada e indexada de elementos, con las siguientes características:

1- **Todos los elementos son del mismo tipo; esto convierte a un arreglo en un tipo de dato homogéneo.**

2- **Los elementos pueden recuperarse en cualquier orden, simplemente indicando la posición que ocupan dentro de la estructura; por este motivo es una estructura indexada.**

3- **La memoria ocupada a lo largo de la ejecución del programa es fija; por esto, es una estructura de datos estática.**

En un arreglo es importante destacar los siguientes conceptos:

El **nombre** del arreglo, el cual está asociado a un área de memoria fija y consecutiva, del tamaño especificado en la declaración.

Un **índice**, que debe pertenecer a un tipo de dato ordinal, el cual permitirá acceder a cada elemento del arreglo. El valor de dicho índice puede verse como el desplazamiento respecto de la posición inicial del arreglo.

Los arreglos pueden ser de distintas **dimensiones**. Esta dimensión indica la cantidad de índices necesarios para acceder a un elemento del arreglo.

6.2.1 Vectores

La forma más simple de arreglo es la denominada **vector** o **arreglo de una dimensión**. Formalmente

Definición

Un vector o arreglo lineal es un tipo de dato arreglo con un índice, es decir, con una dimensión.

La Figura 6.1 presenta en forma gráfica la estructura de un vector de n elementos.

Elem. 1	Elem. 2	Elem. 3	.....	Elem. n-1	Elem. n
posición1	posición2	posición3		posición n-1	posición n

Figura 6.1

En un vector, la manera de hacer referencia a alguno de sus elementos es a través del nombre del arreglo y la posición del elemento dentro del mismo, de la forma:

```
Vector[ posición ]
```

Ejemplo 6.1

Un ejemplo de arreglo de una dimensión, o vector, es la siguiente lista de los primeros 8 números primos:

2	3	5	7	11	13	17	19

Si P es el nombre del vector que contiene la lista mencionada, se puede decir que:

`P[1]=2, P[2]=3, P[3]=5, P[4]=7, P[5]=11, P[6]=13, P[7]=17 y P[8]=19`

Dentro de la memoria de la computadora, cada celda del vector ocupa lugar en forma consecutiva a partir de la dirección inicial de memoria asignada. P, en este caso, representa dicha dirección inicial a partir de la cual se ubican cada una de las celdas que componen el vector. La cantidad de espacio asignada al vector depende del número de celdas que el mismo contiene y del tipo de dato que se asigna a cada una de ellas.

6.2.1.1 Definición de un vector

La declaración de una variable de tipo arreglo, en particular un vector, debe contener tanto el **tipo de elemento** que contendrá esta estructura como el **rango de valores** que puede tomar el **índice**. Con dicho índice queda definida, además, la cantidad de elementos que componen la estructura.

La estructura de definición de un tipo de dato vector tiene la siguiente forma:

```
type nombre_tipo_vector = array [indice] of tipo_de_dato;
```


donde `nombre_tipo_vector` es el identificador que se le da al tipo, `indice` es el rango de valores que puede tomar el índice (en Pascal representa un tipo de dato ordinal, normalmente un subrango de los tipos ordinales predefinidos en el lenguaje), y `tipo_de_dato` es el tipo de dato de los elementos del vector (cualquier tipo de dato que ya exista, tanto predefinido como definido por el usuario).

Junto con la definición del índice queda determinado el número de celdas que ocupa el vector, así como el espacio en memoria necesaria para almacenarlo.

Como ya se ha expresado en capítulos anteriores, una vez definido el tipo (en este caso un arreglo unidimensional o vector) se pueden declarar variables de ese tipo.

```
var nombre_variable_vector : nombre_tipo_vector
```

Una forma alternativa es realizar la declaración de la estructura del vector directamente con la variable:

```
var nombre_variable_vector = array [indice] of tipo_de_dato;
```

Ejemplo 6.2

La declaración de un vector que permita almacenar las temperaturas registradas en una determinada ciudad durante los días del mes de mayo será la siguiente:

```
type tempe = array [1..31] of real;

var temperatura : temp;
```

El tipo vector declarado se llama `tempe`, y `temperatura` es el nombre que recibe la variable vector. Este arreglo unidimensional está compuesto por 31 elementos, cada uno de ellos del tipo de datos real. El índice está definido como un subrango del tipo de dato entero, desde el valor entero 1 hasta el valor entero 31, de allí que el vector tenga 31 celdas para almacenar los elementos.

Desde el punto de vista del tamaño de la estructura en memoria, si se considera que cada número real ocupa 4 unidades de la memoria, es decir, 4 bytes (un byte es equivalente al espacio necesario en memoria para guardar un único carácter y será utilizado a fin de tener un parámetro que permita comparar las distintas estructuras), la declaración anterior implica reservar 31 x 4 bytes = 124 bytes.

Nótese que un número real puede ocupar un número diferente de bytes (6, 8, por ejemplo) según la precisión que se desee tener.

Ejemplo 6.3

Se necesita definir un vector que permita determinar la frecuencia con que aparecen los caracteres alfanuméricos de una determinada secuencia. Si la definición fuera correcta, se podría determinar, con dicha estructura de datos, la cantidad de letras 'd' encontradas o la cantidad de símbolos '#'.

```
type frec = array [char] of integer;

var frecuencia : frec;
```

Nótese que en la definición anterior no se utiliza para el índice un subrango de un tipo ya definido, sino que se emplean todos los elementos que componen el tipo de datos carácter. Así, el vector está compuesto por 256 celdas. Además, cada celda permite almacenar un valor de tipo entero correspondiente a la cantidad de caracteres de esa clase encontrados.

Si el tamaño de cada uno de los elementos de la estructura es de 2 bytes (para almacenar un número entero), la estructura completa ocupará 256 x 2 bytes = 512 bytes.

Ejemplo 6.4

En la siguiente definición se utiliza un tipo de dato definido por el usuario para determinar el índice del vector:

```
type tamaño = (Gigante, Grande, Mediano, Chico, Pequeño);
     ven = array[tamaño] of real;

var ventas : ven;
```

El vector ventas, tendrá en este caso cuatro celdas, identificadas por los literales Gigante, Grande, Chico y Pequeño, respectivamente. Además, cada celda podrá almacenar un valor correspondiente al tipo de dato real.

Dado que el tamaño de cada elemento del arreglo es de 4 bytes, la estructura completa ocupará 4 x 4 bytes = 16 bytes de memoria.

6.2.1.2 Asignación de valores a una variable de tipo vector

La asignación de un valor a un elemento de un vector tiene la siguiente forma:

```
vector[ posición ] := valor
```


donde posición debe ser un valor literal o una variable (en ambos casos debe corresponderse con el tipo de índice definido oportunamente para el arreglo). Por otro lado, valor debe ser un valor literal o una variable del tipo de dato que almacena cada celda del vector.

Ejemplo 6.5

Dadas las declaraciones de los ejemplos del apartado anterior:

```
type frec = array [char] of integer;
     tempe = array [1..31] of real;
     tamaño = (Gigante, Grande, Mediano, Chico, Pequeño);
     ven = array[tamaño] of real;

var temperatura : temp;
    frecuencia : frec;
    ventas : ven;
```

las siguientes asignaciones sobre los vectores son válidas:

`frecuencia['a']:= 4;`	A la celda referenciada con el índice carácter 'a' se le asigna el valor entero 4
`frecuencia['&']:= 0;`	A la celda referenciada con el índice carácter '&' se le asigna el valor 0
`c := 'A';`	c es una variable de tipo carácter.
`frecuencia[c]:= 9;`	La celda correspondiente al carácter 'A' recibe el valor 9
`temperatura[1]:= 2.3;`	La temperatura registrada el primer día del mes fue de 2.3 grados.
`i := 5;`	i es una variable de tipo entero.
`temperatura[i]:= 15.3;`	La temperatura registrada el día 5 del mes fue de 15.3 grados.
`ventas[Grande]:=23.4;`	La celda referenciada con el valor Grande toma el contenido 23.4
`aux := Gigante` `ventas[aux]:= 4;`	aux es una variable del tipo tamaño, definido por el usuario. La celda del vector correspondiente a Gigante toma el valor 4.

El siguiente conjunto de operaciones no son válidas sobre los vectores definidos:

`frecuencia['a']:= 4.5;`	Se intenta asignar a la celda un valor real, cuando la misma está definida como entero.
`frecuencia[4] := 0;`	Se intenta referenciar una celda del vector con un valor entero, pero el tipo de índice definido es carácter.
`c:=2;` `frecuencia[c] := 9;`	Con la asignación indicada c es una variable de tipo entero; se está ante un caso similar al anterior.

`i := 5;` `temperatura[i]:= 'v';`	i corresponde a una variable de tipo entero. La temperatura registrada el día 5 del mes debe ser un valor numérico (entero o real), no puede ser un carácter.
`ventas['grande']:= 2;`	Se intenta referenciar una celda del vector con un *string*, cuando los mismos no son de tipo ordinal.
`aux := Gigante`	Es un caso similar al anterior.
`ventas['aux'] := 4;`	No se está utilizando la variable aux sino un *string*.

El siguiente conjunto de operaciones de entrada y salida de información también es válido, aunque en las operaciones de lectura depende del valor ingresado desde teclado:

`Read(frecuencia['a']);`	Se espera leer un valor entero que se asigna a la celda del vector correspondiente al índice 'a'
`Write(frecuencia['H']);`	Se visualiza en pantalla un valor entero correspondiente al contenido de la celda cuyo valor de índice es 'H'.
`Read(temperatura[4]);`	Se espera leer un valor real que se asigna en la celda del vector correspondiente a la temperatura del cuarto día del mes. i := 2;
`Write(temperatura[i]);`	Se visualiza en pantalla un valor real correspondiente al contenido de la segunda celda del vector temperatura. Write(ventas[Grande]); Se visualiza en pantalla un valor real correspondiente a las ventas realizadas de productos grandes.

6.2.1.3 Ejercicios clásicos utilizando vectores

En esta sección se describen una serie de algoritmos que realizan algunas operaciones clásicas sobre vectores.

Ejemplo 6.6

El siguiente algoritmo, que utiliza la definición anterior del vector temperatura, lee la marca térmica de cada día del mes de mayo de 2000, para luego informar en qué días se produjeron temperaturas por encima del promedio del mes.

```
program Temperatura_Mayo;
  type tempe = array [ 1..31 ] of real;

  var i: integer;
      temperatura : arreglo; Tempe;
      suma_temp : real;
      promedio  : real;
```



```
begin
  {obtiene la temperatura registrada cada día del mes de mayo}
  for i := 1  to 31 do
    begin
      readln( temperatura[i] );{se lee la temperatura de i-ésimo día}
        suma_temp := suma_temp + temperatura[i];
    end;
  promedio := suma_temp / 31;
  writeln( 'Temperatura promedio = ', promedio:6:2 );
  {informa en que días se produjeron temperaturas por encima de la media}
  for i := 1 to 31 do
    if temp[i] > promedio then writeln('el dia ', i, ' superó la media');
end.
```

Ejemplo 6.7

Se presenta un algoritmo que lee una secuencia de caracteres que representan letras minúsculas terminada en '.', e informa la cantidad leída para cada uno de dichos caracteres.

```
program Proceso_frase;
  type frec = array [ char ] of integer;

  var cuantos : frec; { en el arreglo almacena la cantidad de caracteres
                        de cada tipo }
      caracter : char; { para leer de teclado y acceder al vector }

  begin
    { Inicializa los 256 elementos del vector de contadores }
    for caracter := chr(0) to chr(255) do
      cuantos[caracter] := 0;

    {lee la frase y cuenta las apariciones de cada caracter}
    read(caracter);
    while caracter <> '.' do
      begin
        {incrementa el contador del caracter correspondiente}
        cuantos[caracter] := cuantos[caracter] + 1;
        read(caracter);
      end;
    {muestra la cantidad de veces que apareció cada caracter}
    for caracter := chr(0) to chr(255) do
      if cuantos[caracter] > 0 {sólo informa los caracteres
                                que aparecieron}
        then writeln('El carácter ', caracter, ' se utlizó ',
                     cuantos[caracter], ' veces');
  end.
```

Ejemplo 6.8

Es posible que un vector contenga en cada una de sus celdas una estructura de datos compuesta. Por ejemplo, es posible definir un arreglo cuyos elementos sean registros. El siguiente algoritmo muestra la forma en que debe manejarse dicha estructura.

Se desea obtener cierta información correspondiente a los empleados de una empresa distribuidora. La distribuidora posee 20 empleados y los datos de cada uno de ellos son: nombre, estado civil, dirección, y edad. Realizar un algoritmo que, una vez leída la información de los empleados informe el nombre y estado civil de todos aquellos que tengan edad superior al promedio del grupo y, además, el nombre y dirección de aquellos cuya edad sea inferior al promedio del grupo. No debe informar nada para aquellos cuya edad sea exactamente igual al promedio.

```
program Empresa_20;
  type empleado = record
         nombre : string[30];
         domicilio : string[20];
         estado : string[15]; {ej:'soltero', 'casado', etc.}
         edad: integer;
       end;

  procedure Leer ( Var e: empleado );
    { Ingresa los datos de un empleado }
    begin
      write('Nombre          : '); readln(e.nombre);
      write('Domicilio       : '); readln(e.domicilio);
      write('Edad            : '); readln(e.edad);
      write('Estado Civil    : '); readln(e.estado);
    end;

  const cant_emples = 20;

  var empresa : array [1..cant_emples] of empleado;
      edad_promedio: real;
      suma_edades: integer;
      nroemple: integer;

  begin
    { lee los empleados y calcula la edad promedio }
    suma_edades := 0;
    for nroemple :=1 to cant_emples do
      begin
        leer( empresa[nroemple] );
        suma_edades := suma_edades + empresa[nroemple].edad;
      end;
    edad_promedio := suma_edades / cant_emples;
    { recorre nuevamente la estructura para informar segun la edad }
```



```
    for nroemple :=1 to cant_emples do
      with empresa[nroemple] do
        begin
          if edad > edad_promedio
            then writeln('Nombre : ', nombre,
                         ' Estado : ', estado)
            else if Edad < Edad_Promedio
                   then writeln('Nombre : ', nombre, ' Domicilio : ',
                                domicilio);
                   { else si la edad es igual al promedio no hay nada
                     que hacer}
        end;
  end.
```

Ejemplo 6.9

Debido a que los arreglos son estructuras estáticas, la dimensión del mismo debe ser establecida *a priori*. Esto lleva a sobredimensionar la estructura en aquellos casos en que inicialmente no se conoce el número exacto de componentes a utilizar. En estas circunstancias, además de manejar la dimensión real del arreglo (aquella definida en la estructura), se manejan variables que administran la dimensión efectiva del arreglo, esto es, la cantidad de celdas que realmente se utilizan.

El módulo definido a continuación recibe un vector de números enteros, junto con su dimensión y, además, un valor entero. El objetivo del mismo es eliminar del vector todos los elementos que coincidan con el valor recibido.

```
const maxi = 500;

type vector = array[1..maxi] of integer;
     dimension = 0..maxi;

  procedure eliminar ( var v : vector;
                       var n : dimension; { cant. real de elementos del vector
                                            se supone que n <= MAXI }
                       buscado : integer); {valor a buscar y eliminar }

    var ind : integer;   { indice del vector }
        aux : integer;   { indice para eliminar un elemento }

    begin
      ind := 1;
      while (ind <= n) do
        begin
          { avanza sobre el vector mientras no encuentre el elemento a
            borrar}
          while (ind <= n) and (v[ind]<>buscado) do
            ind := ind + 1;
          { o lo encontró o se terminó el vector }
```

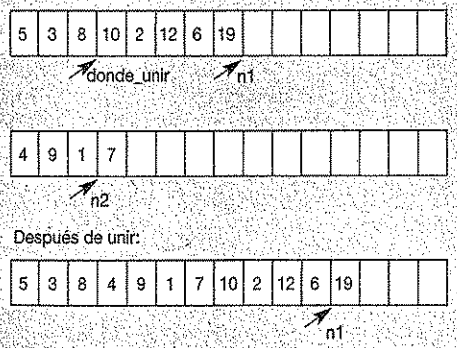
```
            if (ind<=n) then
              begin { lo encontro }
                for aux := ind+1 to n do
                  v[aux-1] := v[aux]; { copia sobre el dato a eliminar
                                        el valor que está en la celda
                                        siguiente del vector}
                n:= n - 1;
              end;
              {ver que no es necesario correr el valor de ind }
          end;
      end;
```

Nótese que la cantidad de elementos asignados en el vector debe ser pasada como parámetro, a fin de distinguir las celdas que realmente están ocupadas. Obviamente el valor de este parámetro debe ser menor o igual que la dimensión real del vector.

Ejemplo 6.10

Se plantea un módulo de un algoritmo que recibe dos vectores de números enteros así como la cantidad de elementos cargados efectivamente en cada uno de ellos. Además, se recibe como parámetro un valor entero, el cual indica la posición a partir de la cual se debe insertar el segundo vector recibido dentro del primero.

Gráficamente, se puede ver de la siguiente manera:



Se tiene como precondición que el vector receptor tiene capacidad para guardar el contenido de ambos.


```
const maxi = 500;

type vector = array[1..maxi] of integer;
     dimension = 0..maxi;

  procedure unir ( var v1 : vector; var n1 : dimension; v2 : vector; n2 :
                   dimension; donde_unir: dimension );

    var ind : integer;

    begin
      { se "corren" hacia atras los elementos de v1 para poder poner los de v2 }
      for ind := n1 downto donde_unir do
        v1[ind+n2] := v1[ind]; { se ponen los elementos de v2 en v1 a
                                 partir de la posición indicada }
      for ind := 1 to n2 do
        v1[donde_unir + ind - 1] := v2[ind];
      n1 := n1 + n2;
    end;
```

6.2.2 Matrices

Los vectores representan la forma más simple de arreglo (unidimensionales). Las matrices representan otro tipo de arreglos denominados bidimensionales.

Definición

Una matriz es un tipo de dato arreglo con dos dimensiones o índices. También puede pensarse en ella como un vector de vectores. Es un grupo de elementos homogéneo, con un orden interno y en el que se necesitan dos índices para referenciar un único elemento de la estructura.

La Figura 6.2 presenta un arreglo bidimensional o matriz de m filas por n columnas.

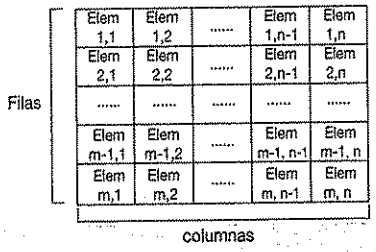


Figura 6.2

De acuerdo a la definición, para referenciar un elemento en particular de una matriz es necesario, además del nombre del arreglo, dos valores de índice que indican la posición en fila y columna. Esto es:

```
Matriz [ posición1, posición2 ]
```

donde `posición1` corresponde a la fila y `posición2` a la columna.

Al igual que ocurre con los vectores, cada celda de la matriz ocupa posiciones consecutivas de memoria, a partir de la dirección inicial asignada. La cantidad de espacio asignado a una matriz depende del tamaño de cada celda y del número de celdas que, en este caso, es igual al producto entre el número de filas por el número de columnas de la matriz. Por ejemplo, una matriz de 3 filas y 6 columnas, donde cada elemento ocupa 4 bytes, ocupará en total 3 x 6 x 4 = 72 bytes.

6.2.2.1 Definición de una matriz en un lenguaje de programación

La declaración de una variable de tipo arreglo bidimensional o matriz contiene el **tipo de elemento** que contiene la estructura (llamado tipo base del arreglo) y la especificación del **tamaño** de la estructura a través de la indicación del tipo al cual pertenece **cada índice**.

La estructura de definición de un tipo de dato matriz tiene la siguiente forma:

```
type nombre_tipo_matriz = array [indice1, indice2] of tipo_de_dato;
```

donde `nombre_tipo_matriz` es el identificador que se le da al tipo, `indice1` determina el rango de valores que puede tomar el índice de **filas** (en Pascal representa un tipo de dato ordinal), `indice2` determina el rango de valores que puede tomar el índice de **columnas** (en Pascal, un tipo ordinal) y no necesariamente es igual o compatible con el índice de filas. Por último, `tipo_de_dato` es el tipo de dato de los elementos de la matriz (cualquier tipo de dato previamente definido, ya sea predefinido por el lenguaje o definido por el usuario). Con la definición de los dos índices queda establecida la cantidad de celdas de la matriz, así como el espacio necesario en la memoria para su almacenamiento.

Como ya se ha expresado, una vez definido el tipo (en este caso un arreglo bidimensional o matriz) se pueden declarar variables de ese tipo.

```
var nombre_variable_matriz : nombre_tipo_matriz
```

Una forma alternativa es realizar la declaración de la estructura de la matriz directamente con la variable:

```
var nombre_variable_matriz = array [indice1, indice2] of
tipo_de_dato;
```


Ejemplo 6.11

La declaración de una matriz que permita almacenar las temperaturas registradas en una determinada ciudad durante el último año ordenadas por cada mes será la siguiente:

```
type tempe = array [1..12, 1..31] of real;

var temperatura : temp;
```

El tipo matriz declarado se llama `tempe`, y `temperatura` es el nombre que recibe la variable matriz. A diferencia del ejemplo con vectores, `temperatura` ahora es una matriz compuesta por 12 filas y 31 columnas, formando así 372 celdas donde cada una puede almacenar un valor real. Note que ambos índices, si bien representan un subrango de los enteros, no son intercambiables. Esto es, primero debe indicarse el índice de mes, para posteriormente indicar el día. En caso de intercambiar los valores puede producirse un error al intentar referenciar un índice de fila superior al límite máximo 12.

Desde el punto de vista del tamaño de la estructura en memoria, si se considera que cada número real ocupa 4 bytes, como se tienen 12 filas y 31 columnas, la estructura completa ocupará 12 x 31 x 4 bytes = 1488 bytes. Obviamente, hay elementos de la matriz que no contienen información útil (por ejemplo, `i=2,j=30`). Sin embargo, son necesarios por el carácter estático de la estructura.

Ejemplo 6.12

En la siguiente definición se utiliza un tipo de dato definido por el usuario para determinar uno de los índices de la matriz:

```
type tamaño = (Gigante, Grande, Chico, Pequeño);
     ven = array[tamaño, 1..31] of real;

var ventas : ven;
```

La matriz ventas podrá almacenar, en este caso, la cantidad de productos (gigante, grande, chico y pequeño) vendida para cada día de un determinado mes.

Si el tamaño de cada elemento es de 4 bytes, el tamaño de la estructura completa será 4 x 31 x 4 bytes = 496 bytes de memoria.

Nótese que la definición de un arreglo, sea cual fuere su dimensión, queda determinada por la palabra clave `array`. El número de dimensiones definidas a continuación determinan si el arreglo es un vector, una matriz u otro. El número máximo de dimensiones es distinto según el lenguaje de programación utilizado.

6.2.2.2 Asignación de valores a una variable de tipo matriz

La asignación de un valor a un elemento de una matriz es similar a la definida anteriormente, salvo que se deben determinar dos valores de índices en este caso:

```
matriz[ posición1, posición2 ] := valor
```

donde `posición1` y `posición2` deben ser un valor literal o una variable. En ambos casos debe corresponderse con el tipo de índice definido oportunamente para la matriz. Por otro lado, el valor debe ser un valor literal o una variable del tipo de dato que almacena cada celda de la matriz.

Ejemplo 6.13

Dadas las declaraciones de los ejemplos del apartado anterior:

```
type tamaño = (Gigante, Grande, Chico, Pequeño);
     ven = array[tamaño, 1..31] of real;
     tempe = array [1..12, 1..31] of real;

var ventas : ven;
    temperatura : temp;
```

las siguientes asignaciones sobre las matrices son válidas:

`temperatura[1,12]:= 21;`	La temperatura registrada el doceavo día del mes de enero fue de 21 grados.
`i := 5;` `temperatura[i,3]:= 15.3;`	i corresponde a una variable de tipo entero. La temperatura registrada el día 3 del mes de mayo fue de 15,3 grados.

`i:=4`
`j:=30`
`Read( temperatura[i,j]);`
	i y j corresponden a variables de tipo entero.
	Se lee desde el teclado la temperatura registrada el día 30 del mes de abril. Este ejemplo tiene dos consideraciones importantes para realizar, la primera consiste en notar que, si se alteran los índices de fila y columna respectivamente, se tendrá un caso erróneo. La segunda consideración tiene que ver con el valor de j, ya que si el mismo fuera 31 se leería la temperatura del 31 de abril; si bien dicho día no existe, al estar definida una celda con dichas coordenadas no se detectará el ejemplo como un error.

`ventas[Grande,31]:=2.4;`	La celda referenciada con el valor Grande para el día 31 toma el contenido 2,4

`aux := Gigante`
`col := 21;`
`Write(ventas[aux, col]);`
	aux es una variable del tipo tamaño definido por el usuario y col es una variable de tipo entero. El contenido de la celda de la matriz determinada por el valor de aux, para índice de fila y el valor de col para el de columna, es presentado en pantalla.

El siguiente conjunto de operaciones no es válido sobre las matrices definidas:

`temperatura[22,12]:=23;`	El valor del primer índice excede el rango definido.
`temperatura[1,'4']:= 15;`	En este caso, el segundo índice es incorrecto debido a que se utiliza un elemento de tipo carácter cuando se espera un elemento de tipo entero.
`temperatura[3,6]:= 'r'`	El tipo de dato al cual pertenecen los elementos de la matriz es el tipo real y en este caso se intenta asignar un valor carácter.

De acuerdo a lo presentado en los ejemplos anteriores, es posible realizar operaciones de lectura y escritura desde y hacia celdas de una matriz.

6.2.2.3 Ejercicios clásicos utilizando matrices

En esta sección se describe una serie de algoritmos que realizan algunas operaciones clásicas sobre matrices.

Ejemplo 6.14

El siguiente algoritmo, que utiliza la definición anterior de la matriz temperatura, lee la marca térmica de cada día del año 2000, para luego informar en qué días se produjeron temperaturas por encima de la media del mes y del año.

```
program Temperaturas;
  type matriz = array [1..12,1..31] of real;

  var temperatura : matriz;
      fila: 1..12;
      col: 1..31;
      tmes,       {totaliza las temperaturas del mes}
      tano,       {totaliza las temperaturas del año}
      mediaMes,   {guarda la temperatura promedio del mes}
      mediaAno : real; {guarda la temperatura promedio del año}

  begin
    tano := 0;
    { se leen para cada mes,las temperaturas de todos los días}
    { se considera que cada mes tiene 31 días}
```

```
    for fila := 1 to 12 do
      begin
        tmes := 0;
        for col := 1 to 31 do
          begin
            readln( temperatura[fila,col] ); {se lee la temperatura de
                                                i-ésimo día}
            tmes := tmes + temperatura[fila,col];
            tano := tano + temperatura[fila,col];
          end;
        mediaMes := tmes / 31;
        {se obtienen los días en los que la temperatura supera la media del mes}
        for col := 1 to 31 do
          if (temperatura[fila,col]> mediaMes)
            then writeln ('El día ', col, ' superó la media del mes ', fila);
      end;
    mediaAno := tano / 365;
    { se obtienen los días en los que la temperatura supera la media
      del año}
    for fila := 1 to 12 do
      for col := 1 to 31 do
        if (temperatura[fila,col] > mediaAno)
          then writeln ('El día ', col, ' del mes ', fila, ' superó la
                          media del año');
end.
```

Ejemplo 6.15

Se presenta un algoritmo que utiliza un módulo que recibe dos matrices de igual dimensión efectiva, calcula la suma entre ambas y retorna la matriz resultado.

```
program sumamatriz;
  const  fila = 4;
         columna = 3;

  type matriz = array [1..fila, 1..columna] of integer;

  var  i,
       j : integer;
       mat1,
       mat2,
       suma : matriz;
```



```
  procedure Suma_Mat (matriz1, matriz2: matriz; var MatrizSuma : matriz );
    var fil : 1..fila;
        col: 1..columna;
    begin
      { suma los elementos de las matrices recibidas recorriéndolas por fila }
      for fil := 1 to fila do
        for col := 1 to columna do
          MatrizSuma [fil,col]:= matriz1 [fil,col] + matriz2 [fil,col];
    end;

  begin
    {Se ingresan los datos de la matriz mat1}
    for i := 1 to fila do
      for j := 1 to columna do
        readln (mat1[i,j]);

      {Se ingresan los datos de la matriz mat2}
      for i := 1 to fila do
        for j := 1 to columna do
          readln (mat2[i,j]);

      Suma_Mat ( mat1, mat2, suma );

      {Se muestra la matriz resultado de la suma}
      for i := 1 to fila do
        for j := 1 to columna do
          write ( suma[i,j] );
  end.
```

Ejemplo 6.16

En este ejemplo se construye un módulo que, a partir de una matriz de números enteros, genera un vector donde cada elemento del mismo es el resultado de la suma de los elementos de cada fila de la matriz.

```
program sumafila;
  const fila = 4;
        columna = 3;

  type matriz = array [1..fila, 1..columna] of integer;
       vector = array [1..fila] of integer;

  var i,
      j : integer;
```

```
    mat : matriz;
    VecGen : vector;
    suma : integer;

begin
  {Se ingresan los datos de la matriz mat}
  for i := 1 to fila do
    for j := 1 to columna do
      readln (mat[i,j]);

  { suma los datos de la fila i-ésima y guarda el resultado como }
  { componente i-ésima de VecGen }
  for i := 1 to fila do
    begin
      suma := 0;
      for j := 1 to columna do suma := suma + mat [i,j];
      VecGen [i] := suma;
    end;

  {Se muestra el vector generado }
  for i := 1 to fila do write ( VecGen[i] );

end.
```

6.2.3 Arreglos n-dimensionales

La cantidad de dimensiones permitidas para un arreglo está acotada por el lenguaje de programación y, básicamente, por el espacio que la estructura ocupa en la memoria. Si se tiene un arreglo de 4 dimensiones con 10 elementos por cada dimensión se están manipulando 10.000 celdas de memoria. Si suponemos que en dichas celdas se almacenarán un tipo de dato *string* de 20 caracteres, estamos utilizando 200.000 bytes de memoria. Analice el lector el espacio que se utilizaría si se incrementa el número de elementos por dimensión o si se almacena un registro de 100 bytes en cada celda.

Como se expresó anteriormente, los arreglos se definen siempre utilizando la misma metodología, solo hay diferencia en la definición de las dimensiones. Así, la siguiente declaración:

```
type tensor = array [1980..1990,1..12, 1..31] of integer;

var arreglo : tensor;
```

define un arreglo de tres dimensiones, que recibe el nombre de **tensor**. Los índices de cada dimensión están determinados por sub-rangos de tipo de dato entero. La operación con esta variable es similar a lo explicado oportunamente para matrices, y debe ponerse sumo cuidado en

no alterar el orden de los índices. Para utilizar la celda correspondiente al 13 de marzo de 1986 bastará con hacer referencia a la misma, de la siguiente forma:

```
arreglo[ 1986, 3, 13].
```

Si el tamaño de cada uno de los elementos de la estructura es de 2 bytes, como se tienen 11 estructuras de 12 filas y 31 columnas cada una, la estructura completa ocupará 11 x 12 x 31 x 2 bytes = 8184 bytes.

Ejemplo 6.17

Supóngase que se quiere conocer la temperatura promedio de los años 1995, 1996 y 1997 y averiguar qué días de este período superaron dicho promedio.

```
program Temperaturas_tensor;
  type tensor = array [1995..1997,1..12,1..31] of real;

  var temperatura : tensor;
      est: 1995..1997;
      fila: 1..12;
      col: 1..31;
      anio3, { totaliza las temperaturas de los últimos tres años}
      mediaAno3: real;   {guarda temperatura promedio de los tres años}

  begin
    anio3 := 0;
    { se leen para cada año, las temperaturas de todos los días de cada
     mes considerando que cada mes tiene 31 días }
    for est := 1995 to 1997 do
      for fila := 1 to 12 do
        for col := 1 to 31 do
          begin
            readln (temperatura[est,fila,col]);
            anio3 := anio3 + temperatura[est,fila,col];
          end;
    {cálculo del promedio suponiendo que cada año tiene 365 días}
    mediaAno3 := anio3 / 1095;

    {se obtienen los días en que se supera la media de los tres años}
    for est := 1995 to 1997 do
      for fila := 1 to 12 do
        for col := 1 to 31 do
          if ( temperatura[est,fila,col] > mediaAno3)
            then writeln ('El ', fila, '/', col, '/', est, ' superó
                          el promedio');
  end.
```

En general, la definición de un arreglo n-dimensional tiene la forma:

```
type nombre_tipo_arreglo_ndim = array [indice1, indice2,....., indi-
cen] of tipo_de_dato;
```

donde cada índice es un subrango de algún tipo ordinal y `tipo_de_dato` representa cualquier tipo de dato ya definido. Una vez definido el tipo pueden definirse variables del mismo.

6.2.4 Arreglos como parámetros

Un arreglo puede ser enviado como parámetro a un módulo o puede recibirse como respuesta de un determinado proceso. La forma en que los arreglos son enviados o recibidos como parámetros coincide con las definiciones previas efectuadas. Esto es, un módulo que necesita información contenida en un arreglo solo como dato de entrada, recibirá el parámetro por valor. Por el contrario, necesitará recibirlo por referencia si debe retornar el arreglo modificado.

Desde el punto de vista de la eficiencia en la utilización de recursos no siempre es aconsejable que las estructuras de datos arreglo sean enviadas como parámetros por valor. Se debe tener en cuenta la definición presentada oportunamente para dichos parámetros, la cual indicaba que el contenido de la variable que se envía es copiado sobre la variable del módulo que recibe el dato. De esta forma, la memoria de la computadora está siendo ocupada simultáneamente por dos variables compuestas distintas con igual contenido.

Si se está trabajando con variables simples, o eventualmente con un elemento del tipo de dato registro, esta duplicación en el uso de la memoria es despreciable. Por ejemplo, si una variable de tipo de dato entero ocupa 2 bytes, tenerla por duplicado significa utilizar 4 bytes. Eventualmente, un registro de 150 bytes repetido involucra utilizar 300 bytes.

¿Qué sucede si el tipo de dato que se envía como parámetro es un arreglo? La cantidad de memoria involucrada en la estructura depende directamente del número de celdas que lo componen. Un arreglo bidimensional, con 1000 filas y 1000 columnas significa 1.000.000 de celdas, si cada una de ellas ocupara 8 bytes (por ejemplo, un dato número real para algunos lenguajes), dicho arreglo ocuparía 8.000.000 bytes en memoria (aproximadamente 8 Mb), al pasarlo como valor, se repite en la memoria la estructura, ocupando de esta forma 16 Mb. Esto puede generar problemas con la memoria disponible para los datos.

De acuerdo a esto, no siempre es conveniente, por razones de eficiencia en la utilización de un recurso como la memoria, enviar a un módulo un tipo de dato arreglo como parámetro por valor. En estos casos conviene utilizar un parámetro por referencia, obviamente tomando los recaudos para no modificar el arreglo dentro del módulo invocado, ya que esto tendría un efecto lateral *(side effect)* sobre los datos en el módulo que llama.

6.3 Comparación de estructuras de datos arreglo con pilas y colas

Los arreglos, junto con las pilas y las colas son, hasta el momento las únicas estructuras de datos homogéneas y compuestas que se han desarrollado en este libro. Los arreglos son estructuras estáticas, a diferencia de pilas y colas que son estructuras dinámicas. En este apartado se analizan estas estructuras observando su organización interna, viendo cómo afecta dicha organización en la eficiencia de cada una de ellas.

Los arreglos son estructuras estáticas y, por este motivo, la cantidad de celdas que los componen debe estar establecida de antemano en la definición de la estructura, dado que el espacio en memoria les es reservado al principio del algoritmo. Si se utiliza esta estructura de datos en problemas donde *a priori* no es bien conocido el número de celdas a utilizar, se debe sobredimensionar el arreglo para luego trabajar con la dimensión efectiva del mismo.

Las pilas y colas, por el contrario, son estructuras dinámicas y no tienen el inconveniente que presentan los arreglos respecto de la utilización de espacio en memoria, el cual es asignado o liberado en función de las necesidades del problema.

Un principio aceptable podría indicar que, si en un problema se puede determinar o acotar la cantidad de espacio de almacenamiento de datos necesario, es posible utilizar un arreglo como estructura de datos y, en caso contrario, se debería utilizar pilas o colas.

Si bien desde un punto de vista de aprovechamiento de la memoria es más conveniente la utilización de estructuras dinámicas, estas presentan un problema subyacente: el acceso a las mismas. Si se debe recuperar, por ejemplo, el cuarto elemento de un vector, basta con acceder directamente a la cuarta celda del mismo, y así para cualquier celda del arreglo. Por el contrario, el mismo problema utilizando pilas o colas presenta mayores inconvenientes: se necesita un algoritmo y estructuras de datos auxiliares para llevarlo a cabo. En el siguiente ejemplo se desarrollan tres funciones: la primera retorna el i-ésimo elemento de un vector, la segunda lo hace recibiendo una estructura de tipo pila, mientras que la tercera utiliza una cola (la sintaxis utilizada no es exactamente la de Pascal).

Ejemplo 6.18

```
program Compara_estructuras;
  type vector = array[1..50] of tipoElem;

  var pila : stack;
      cola : queue;
      vec: vector;
      elem: tipoElem;
      nro: integer;
```

```
function Ver_1 (datos : vector; posicion : integer) : tipoElem;
  { Retorna el elemento del vector que se encuentra en el lugar posición }
  begin
    Ver_1 := datos[posicion];
  end;

function Ver_2 (datos : stack; posicion : integer) : tipoElem ;
  { Retorna el elemento de la pila que se encuentra en el lugar posición }
  var aux : stack;
      i : integer;
      elem : tipoElem;
  begin
    st_Create(aux);
    { saca los primeros elementos y los guarda en una estructura auxiliar }
    {supone que posición <= cantidad de elementos de la pila}
    for i := 1 to posicion-1 do
      begin
        st_Pop(Datos,elem);
        st_Push(aux, elem);
      end;
    Ver_2 := st_Top(datos);  { no lo desapila }
    {restaura los elementos que se encuentran arriba en la pila}
    for i := 1 to posicion-1 do
      begin
        st_Pop(aux,elem);
        st_Push(datos, elem);
      end;
  end;

function Ver_3 (datos : queue; posicion : integer) : tipoElem ;
  { Retorna el elemento de la cola que se encuentra en el lugar posición}
  var i : integer;
      elem : tipoElem;
  begin
    {saca los primeros elementos y los guarda al final de la cola}
    for i:= 1 to posicion do
      begin
        q_Pop(datos,elem);
        q_Push(datos, elem);
      end;
    Ver_3 := elem;
    {hace pasar los elementos de manera de restaurar el orden inicial}
    for i := posicion+1 to q_length(datos) do
      begin
        q_Pop(datos,elem);

        q_Push(datos, elem);
      end;
  end;
```



```
begin
  st_Create( pila );
  q_Create( cola );
  writeln ('Ingrese una secuencia de 10 números enteros ');
  for nro := 1 to 10 do
    begin
      readln (elem);
      st_Push( pila, elem);
      q_Push( cola, elem );
      vec[nro] := elem;
    end;
  writeln('4to. elemento del arreglo = ',Ver_1(vec,4));
  writeln('4to. elemento de la pila  = ',Ver_2(pila,4));
  writeln('4to. elemento de la cola  = ',Ver_3(cola,4));

  { Muestra el contenido de las estructuras }
  write('Pila   = ');
  while not St_empty(Pila) do
    begin
      st_pop(pila, elem);
      write(elem,'  ');
    end;
  writeln;

  write('Cola   = ');
  while not q_empty(cola) do
    begin
      q_pop(cola, elem);
      write(elem,'  ');
    end;
  writeln;

  write('Vector = ');
  for nro := 1 to 10 do write(Vec[nro], '  ');

end.
```

6.3.1 Resultado de la ejecución del algoritmo anterior

Ingrese una secuencia de 10 números enteros:

56 67 78 89 90 11 12 13 14 15.

4to. elemento del arreglo = 89

4to. elemento de la pila = 12

4to. elemento de la cola = 89

Pila = 15 14 13 12 11 90 89 78 67 56

Cola = 56 67 78 89 90 11 12 13 14 15

Vector = 56 67 78 89 90 11 12 13 14 15

El programa anterior genera tres estructuras de 10 elementos y luego busca el cuarto elemento de cada una. Con solo ver el código es posible concluir que el uso de una pila o una cola para resolver este problema implica más trabajo que en el caso del vector. Sin embargo, este último no es capaz de contener más de 50 elementos.

Analizando la cantidad de accesos a la estructura correspondiente en cada función y considerando N elementos en cada caso, el pretender conocer el valor de i-ésimo el elemento de la estructura se muestra en la Tabla 6.1.

Función	Cantidad de accesos (lecturas y escrituras)
Ver_1	1 única lectura
Ver_2	Desapilado de los elementos iniciales = (i-1) lecturas + (i-1) escrituras en una estructura auxiliar. Obtención del valor = 1 lectura usando St_Top que no lo desapila Restauración de los valores iniciales = (i-1) lecturas de la estructura auxiliar + (i-1) escrituras en la pila. Total = 2*(i-1) + 1 + 2*(i-1) = 2 * (i-1) + 1
Ver_3	Desencolado de los elementos iniciales = (i-1) lecturas + (i-1) escrituras en la misma cola (los coloca al final). Obtención del valor = 1 lectura + 1 escritura en la misma estructura. Restauración de la posición inicial de los elementos = (N-i) lecturas de la cola + (N-i) escrituras. Total = 2*(i-1) + 2 + 2 * (N-i) = 2 * (i - 1 + 1 + N - i) = 2 * N

Tabla 6.1

Conclusiones

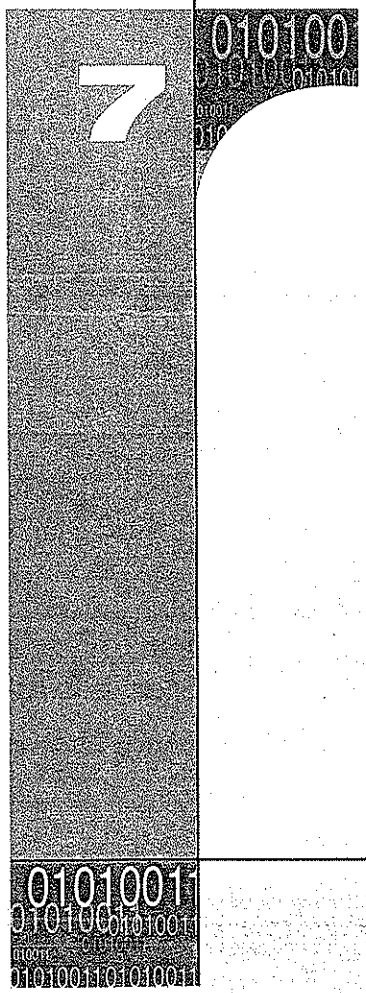
La estructura de datos arreglo es la más utilizada de las estructuras de datos compuestas, por su facilidad de acceso y la simplicidad de sus operaciones. Está presente en todos los lenguajes de programación.

Los mayores inconvenientes en el uso de arreglos se relacionan con sus características **estáticas** que obligan, en ocasiones, a sobredimensionar la memoria requerida.

Una vez más se le insiste al lector en que, dado el problema del mundo real y su modelización abstracta, deben elegirse las estructuras de datos adecuadas para la aplicación.

Ejercicios

1. Realizar un algoritmo que lea una sucesión de caracteres terminada en punto. El algoritmo debe informar para cada letra minúscula la cantidad leída. Tener en cuenta que se puede leer cualquier tipo de caracter.
2. Desarrolle un problema que lea un conjunto de datos que representan productos y sus características (nombre del producto, presentación, *stock* y costo). La cantidad de productos manejada estará acotada a 250. El algoritmo debe informar el nombre del producto y presentación del mismo y si el *stock* se encuentra por debajo del promedio del lote. Además, se debe decrementar en un 5% el costo de aquellos que estén por encima del promedio del grupo.
3. Realizar un procedimiento que reciba un vector, su dimensión efectiva y dos posiciones dentro del vector. El procedimiento debe retornar el vector donde todos los elementos existentes entre las posiciones recibidas han sido eliminados.
4. Definir las estructuras necesarias y luego desarrollar un algoritmo que permita leer una matriz de 5 filas por 5 columnas triangular superior.
5. Realizar un módulo que reciba dos matrices junto con sus dimensiones y retorne el producto entre ambas
6. Realizar un módulo que permita leer y guardar la información asociada con el siguiente problema: un cierto grupo estadístico realiza una encuesta a personas, categorizando a las mismas por edad (rangos: 0..19, 20..29, 30..39, 40..49, 50 o +), nivel socio-económico (que puede ser A, B o C) y estudios realizados (universitario, secundario, primario, ninguno).



Capítulo 7

Recursividad

7

Recursividad

Objetivo

En este capítulo se define el concepto de recursividad. Se plantea su uso para formular soluciones a problemas que por sus características requieren diseñar algoritmos que deben llamarse a sí mismos.

Se propone, además, una estrategia de implementación de problemas recursivos, luego se hace un análisis comparativo entre soluciones iterativas clásicas y las soluciones que utilizan la recursividad.

7.1 Recursividad

La recursividad es una herramienta que permite expresar la resolución de problemas evolutivos, donde es posible que un módulo de software se invoque a sí mismo en la solución del problema. (Garland, 1986), (Hellman, 1991), (Aho, 1988).

Esta técnica permite diseñar la solución a un problema P por resolución de instancias más pequeñas P_1, P_2, P_3, ...P_n del mismo problema P. Si bien esta metodología puede parecerse a la metodología de diseño *Top Down*, donde un problema complejo se divide en otros más simples, usando la recursión, el problema P_i (es una instancia más pequeña) es de la misma naturaleza que el problema original P, pero en algún sentido es más simple que P.

Definición

Cuando se obtienen soluciones a problemas en los que una función o procedimiento se llama a sí mismo para resolver el problema, se tienen subprogramas recursivos

Otro tipo de recursión es la referente a los tipos de datos que se definen en función de sí mismos como por ejemplo:

La definición recursiva de los números naturales:

a) 1 es un número natural

b) el número siguiente a un número natural es un natural

7.1.1 Aspectos a tener en cuenta en una solución recursiva

Suponiendo que se quiere buscar el número telefónico de una persona en la guía. Una forma de resolverlo sería:

Buscar la persona en la guía:

Código

```
si la guía contiene una sola página
   entonces  Buscar secuencialmente la persona dentro de la página
   sino
      abrir la guía a la mitad
      determinar en que mitad está la persona
      si la persona está en la 1era mitad
         entonces Buscar en la 1era mitad de la guía
         sino Buscar en la 2da mitad de la guía
fin
```


En esta solución hay algunas tareas poco claras, por ejemplo, cómo buscar una persona dentro de la guía, cómo hallar la mitad de la guía, etc. Estas cuestiones serán tratadas posteriormente, por ahora se examina solo la estrategia.

Se ha reducido el problema de buscar la persona en toda la guía, a buscarla solo en una mitad. Además, se puede observar que en cada instancia se trabaja con la mitad de la guía de la instancia anterior. (Figura 7.1)

Si, por ejemplo, la guía tuviera inicialmente 200 páginas, luego de seis invocaciones "la guia" sobre la que se busca, estaría reducida a lo sumo a 4 páginas. Luego de ocho invocaciones se tendría el caso degenerado (una página) y finalizaría la búsqueda.

Se pueden realizar tres observaciones importantes a este problema:

- Una vez dividida la guía, y determinada la parte que contiene a la persona, el método de búsqueda a aplicar sobre esa mitad es el mismo que el empleado para la guía completa.
- La mitad de la guía donde no está el dato se descarta, lo cual significa una reducción del espacio del problema.
- Hay un caso especial que se resuelve de una manera diferente que el resto, y sucede cuando la guía queda reducida a una sola página (luego de varias subdivisiones).

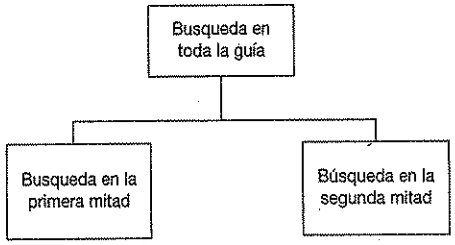


Figura 7.1

En este punto el problema es tan simple que puede resolverse de una manera secuencial. Esta situación se denomina caso base o degenerado.

Como se puede notar, el método aplicado consiste en dividir y dividir la guía hasta obtener una sola página donde podría estar la persona buscada, es decir, la división se hace hasta que el caso degenerado sea alcanzado. Esta estrategia es inherente a cualquier solución recursiva.

Una forma más rigurosa de escribir el algoritmo planteado podría ser:

Código

```
Buscar ( guía, persona ) { la guía se actualiza (reduce) en cada instancia }
if la guía contiene una sola página
   then  buscar la persona dentro de la página ( secuencial )
   else
      abrir la guía a la mitad
      determinar en que mitad está la persona
            if la persona está en la 1era mitad
                then buscar (1era mitad de la guía, persona)
         else buscar (2da mitad de la guía, persona)
end
```

Al escribir la solución como un procedimiento permite hacer algunas observaciones importantes:

- Una de las acciones del procedimiento es **llamarse a sí mismo.** Es decir, el procedimiento `Buscar` es llamado desde el mismo procedimiento `Buscar`. Esta acción es lo que hace que el procedimiento sea recursivo. La estrategia es dividir la guía por la mitad y volver a aplicar la misma idea sobre la mitad seleccionada.
- Cada llamada al procedimiento `Buscar` realizada desde dentro del procedimiento `Buscar` pasa como parámetro la mitad de la guía, "guia actual". Es decir, cada llamada sucesiva a `Buscar` (guía, persona) reduce el tamaño de la guía a la mitad. El problema de `Buscar` es resuelto por otro problema de `Buscar` de igual naturaleza pero de menor tamaño.
- Hay una instancia de `Buscar` que se resuelve de manera distinta a los demás. Este es el caso en que la guía contiene una sola página y se lo denomina caso degenerado como se señaló antes. Cuando esta situación se alcanza la recursividad termina y el problema se resuelve directamente. Es importante destacar que la **reducción en el tamaño del problema garantiza que el caso degenerado sea alcanzado.**

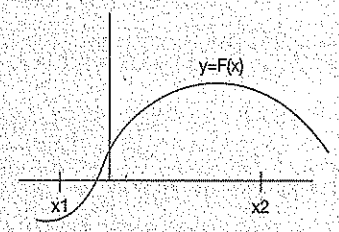
7.1.2 Casos de estudio

Para construir una solución recursiva para un problema cualquiera se deben hacer las siguientes preguntas:

- ¿Cómo se puede definir el problema en términos de un problema más simple de la misma naturaleza?
- ¿Cómo será disminuido el tamaño del problema en cada llamado recursivo?
- ¿Qué instancia del problema servirá como caso degenerado?

Ejemplo 7.1

Sea f(x) una función como muestra la figura, dados dos valores de un intervalo [x1,x2] (sobre el eje X) y un valor delta, sabiendo que la función tiene una raíz en el intervalo dado, se pide obtener un valor que se aproxime a la raíz con un error máximo delta.



Una solución posible podría ser acercarse al cero de la función a partir de "disminuir" el intervalo por sucesivas particiones del mismo. En cada partición se queda con el intervalo en el cual la evaluación de la función en los extremos del intervalo tiene distinto signo.

```
Program Raíz;
  const a = 1;            {punto del intervalo}
        b = 20;           {punto del intervalo}
        delta = 1 E -6    {error permitido}

  var z: real;

  Function F ( x: real ): real;
    {este módulo calcula el valor de la función en el punto x}
    begin
      F := x * x - 5;
    end;


  Function Cero ( x1, x2, delta: real ): real; {solución iterativa}
    {supone F continua y signo F(x1) <> F(x2)}
    {devuelve un valor en el intervalo que aproxima en Delta a la raíz}
    var x: real;
        s: boolean;

  Function Signo ( r: real ): boolean;
    {devuelve verdadero si el signo es positivo y falso si es negativo}
    begin
      if r >= o
        then Signo := True
        else Signo := False;
    end;
```

```
begin {Cero}
   s := Signo(F(x1));
   repeat
      x := (x1 + x2 ) / 2;
      if s = Signo(F(x))
         then x1 := x
         else x2 := x
   until ( Abs(x2 - x1) < delta );
   cero := ( x1 + x2 ) / 2;
end;

begin  {principal}
   z := Cero ( a, b, delta );
   writeln ( 'La raíz de x*x - 5 en [1,20]es: ', z );
end.
```

Si se analiza este esquema de solución iterativa, se ve que:

- Si se alcance el objetivo a Terminar (que significa `Abs(x2-x1) < delta`)
- De lo contrario se debe reiterar el cálculo aproximándose al objetivo.

En el esquema mostrado se observa que se puede plantear una solución que se llame a sí misma, dado que en cada iteración no se modifica la naturaleza del problema original, sino que se lo simplifica "achicando" el intervalo de estudio.

Analizando la solución anterior se reemplaza `Repeat...Until` por una sucesión de invocaciones a la misma función.

Código

```
Function Cero_Recursiva( x1,x2, delta: real ): real;  { solución recursiva }

  var  puntoMedio: real;

  Function Signo ( r: real ): boolean;
    {devuelve verdadero si el signo es positivo y falso si es negativo}

    begin
      if r >= o
         then Signo := True
         else Signo := False;
    end;

  begin {Cero}
    puntoMedio := (x1 + x2) / 2;
    if ( Abs (puntoMedio - x1 ) < delta )
```



```
        then CeroRecursiva := PuntoMedio
        else if ( Signo(F(x1) = Signo(F(puntoMedio))
                then
                   Cero_Recursiva := Cero_Recursiva (puntoMedio, x2, delta)
                else
                   Cero_Recursiva := Cero_Recursiva (x1, puntoMedio, delta)
  end;
```

Se puede analizar que, tanto la solución iterativa como la solución recursiva conducen a un resultado correcto. Sin embargo, se debe tener en cuenta, que existe una diferencia en el manejo de memoria en ambos enfoques. En general, la naturaleza dinámica de las soluciones recursivas con instancias "pendientes" hasta alcanzar el caso degenerado requieren más memoria y, si bien consumen más tiempo, tanto la lectura del algoritmo como su mantenimiento, es más sencillo.

Para poder aplicar este análisis se propone el cálculo del factorial de un número mayor o igual que 0. Matemáticamente, el factorial de un número (n) se define de la siguiente manera:

Factorial (n) = n * (n-1) * (n-2) * .....* 1
Factorial (0) = 1

Código

```
Function Factorial ( n: integer): real;  { solución iterativa }
  var
     i: integer;   { indice de la repetición }
     fact: real;   { contiene el cálculo parcial }
  begin
     fact := 1;
     for i:= n downto 1 do
        fact := fact * i;
     Factorial := fact
  end;
```

Si se observa la definición matemática surge que la función factorial se puede redefinir en términos de la función Factorial de un número menor:

Factorial(n) = n * Factorial (n-1) = n * (n - 1) * Factorial (n-2) = ...

Esta misma idea puede usarse para definir en general, el Factorial(n-i) en función del Factorial(n-i +1) y así sucesivamente. Finalmente se llega al Factorial(0) cuyo valor es 1 por definición y se lo puede considerar como el caso degenerado.

Código

```
function Factorial ( n: integer ): Real;     {solución recursiva }
  begin
    if ( n <= 1 )
       then Factorial := 1
       else Factorial := n * Factorial ( n - 1 )
  end;
```

Para entender mejor cómo se ejecutan los procesos recursivos es importante analizar el comportamiento de la pila de activación. Básicamente, en un procesador existe un módulo activo (instrucciones y datos) y cuando este llama a otro módulo que **toma** el control del procesador, el nuevo módulo se "apila" en memoria con sus instrucciones y datos locales.

Se define a la "pila de activación" como una estructura de datos que se comporta de la siguiente manera:

- Tiene un tope (*top*) y un fondo (*bottom*)
- Cada elemento es código más datos.
- Los elementos pueden sacarse solo por el tope. El único elemento posible de sacar es el que está en el tope.

7.2 Ejecución de un programa y la pila de activación

La pila de activación presenta el comportamiento de la estructura pila que se acaba de analizar.

Cuando un programa comienza su ejecución, se crea la pila de activación. De ahora en adelante, cada elemento que se agregue a la pila ocupará un segmento de la memoria, el cual se denominará *frame*. Por lo tanto, el primer elemento que se pone en la pila es el *frame* correspondiente al programa principal. En ese *frame* se guardan dos clases de datos: todos los datos declarados en la sección de declaración del programa principal y una variable que contiene la línea de código que se está ejecutando (Contador de programa).

Cada vez que se invoca a un procedimiento o función, se asigna un nuevo *frame* para ese procedimiento o función y permanece en la pila hasta que termine su ejecución. En ese momento el espacio de memoria asignado se libera y el programa o módulo de programa que había realizado la invocación vuelve a retomar el control (pero está en el tope de la pila).

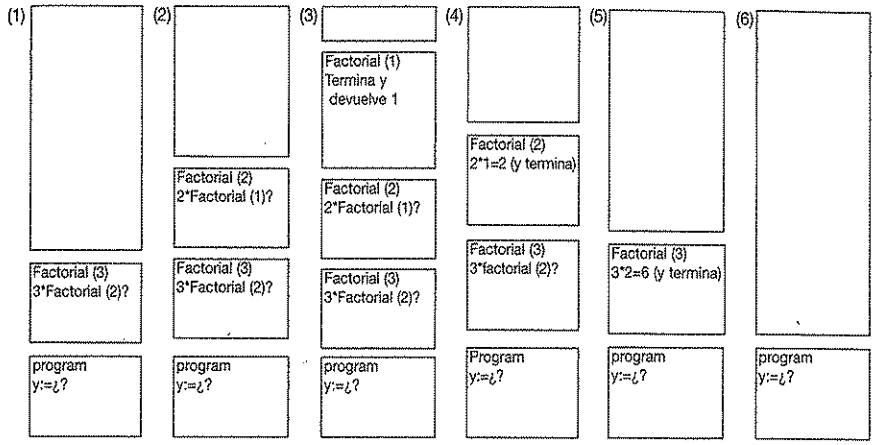


Figura 7.2

A continuación se sigue la ejecución de la función, por ejemplo, para n = 3. Para ello, se utiliza la pila de activación ya presentada.

Suponiendo que la función es invocada desde el programa principal con:

```
y := Factorial ( 3 );
```

En la pila de activación, con cada invocación a la función se hace una copia del área de datos para esa función, por lo tanto, en la primera la pila toma el aspecto de la Figura 7.2.

Se puede observar que la invocación de Factorial con parámetro 1, se resuelve sin necesidad de una nueva llamada recursiva, por lo tanto, la invocación retorna el valor 1 (caso degenerado), cuando esta función termina su ejecución se libera el espacio ocupado en la memoria y el control retorna a Factorial con n=2, que ahora puede calcular el factorial de 2, que cuando termina su ejecución, libera el espacio de la memoria ocupado y retorna el control a Factorial con n=3. Recién ahora puede calcular el factorial de 3, termina su ejecución, libera el espacio de memoria ocupado y retorna el control al programa principal y asigna a la variable y, el valor 6.

En el tercer paso se observa que existe una máxima ocupación de la memoria (hay 4 *frames* en la pila de activación).

Esta ocupación puede ser crítica para problemas más exigentes. Por ejemplo, si n=500 con este método habría un máximo de 500 *frames* activos y si cada *frame* requiere $2x10^4$ bytes se requieren de 10 Mbyte para la pila de activación.

Ejemplo 7.2

1) Para el cálculo de la potencia de un número X^n, se puede pensar en un definición iterativa de la siguiente manera:

$X^n = X * X * \ldots\ldots\ldots * X$ (X se multiplica n veces)

```
Function Potencia (x: real; n: integer): real;
   var i: integer;
       prod: real;
   begin
     prod := x;
     for i := 2 to n do
         prod := prod * x
     Potencia := prod;
   end;
```

o en una definición recursiva de la siguiente manera:

$X^n = X * X^{n-1}$

```
Function Potencia (x: real; n: integer): real; {solución recursiva}
   begin
     if n = 0
       then Potencia := 1
       else Potencia := x * Potencia (x, n-1)
   end;
```

Nuevamente, se visualiza en forma clara la solución recursiva, aunque si N es grande se requerirá una memoria y un tiempo mucho mayor que en la solución iterativa.

2) Para imprimir los dígitos de un número entero no negativo de derecha a izquierda. Dado, por ejemplo, el número 3251, obtener como resultado: 1, 5, 2, 3.

```
Procedure Reverso ( n: integer );  { solución iterativa }
   begin
     write ( n mod 10 );
     while ( n div 10 <> 0 ) do
     begin
                 { obtiene un nuevo valor de n (cociente) }
                 n := n div 10;
                 write ( n mod 10 );
     end;
   end;

Procedure Reverso ( n: integer );  { solución recursiva }
   begin
     write ( n mod 10 );
     n := n div 10
     if  n <> 0
       then Reverso( n )
   end;
```


A través de los ejemplos planteados se puede observar que una forma general para los algoritmos recursivos responde al siguiente esquema:

```
Si ( alguna condición de terminación )
   entonces
     tomar alguna acción final y terminar
   sino tomar acción que se acerque a la solución y hacer una
     llamada recursiva
```

Una solución recursiva debe asegurar el caso degenerado.

Conclusiones

La recursividad es un arma poderosa para expresar la solución en forma sintética y clara de aquellos problemas donde un módulo debe invocarse a sí mismo.

Algunos problemas pueden resolverse con la misma facilidad usando tanto recursividad como iteración.

En general, para un mismo algoritmo la recursividad permite una expresión más clara y sintética lo que simplifica la comprensión y el mantenimiento del mismo. A su vez, las soluciones recursivas son normalmente menos eficientes, ya que consumen más tiempo y memoria.

La recursividad es apropiada, también, cuando los datos involucrados en el problema están organizados en estructuras que pueden definirse recursivamente, como listas, árboles, etc. tal como se verá posteriormente.

En general, para poder manejar la recursividad los lenguajes de programación (y el sistema operativo asociado) deben tener un manejo dinámico de memoria y un control de activación de instancias múltiples de los módulos recursivos.

Ejercicios

1. Resolver en forma iterativa y recursiva el problema de encontrar una raíz de la función $x^3 - 2x^2 - 4$ en el intervalo [2,3], con error menor a 0,01.
2. Resolver en forma iterativa y recursiva la integral de la función $x^3 - 2x^2 + 4$ en el intervalo [1,4] con una aproximación de 0,01. Como sugerencia, utilizar el algoritmo de Simpson.
3. Resolver en forma recursiva el recorrido del robot por toda la Ciudad, dejando un caramelo en cada esquina.
4. Analizar un algoritmo para la obtención de los números primos y reescribirlo recursivamente para obtener los 20 primeros números primos.
5. Dada una cola que representa una palabra, determinar si la misma es un palíndromo. Implementar una solución recursiva y una no recursiva.
6. Escribir un programa en el que dada una serie de palabras imprima todas las sub-secuencias posibles, en el orden en que aparecen.

Ejemplo: Dada la serie casa árbol gol rojo sol
Imprima casa árbol gol rojo sol
árbol gol rojo sol
gol rojo sol
rojo sol
sol

Implemente una solución recursiva y una no recursiva.

7. Escribir un programa en el que dadas dos series, una de letras y otra de números, las dos de igual longitud, imprima los posibles pares donde el primer elemento es una letra y el segundo un numero.

Ejemplo: Dada una serie A B C y otra 1 2 8
Imprima A1 A2 A8
B1 B2 B8
C1 C2 C8

Implemente una solución recursiva y una no recursiva.

8. Calcular el máximo común divisor entre dos enteros positivos utilizando un algoritmo recursivo.

9. Verificar si una expresión aritmética (que se recibe en una cola) que contiene paréntesis, corchetes y llaves, está balanceada. La asociatividad está dada por paréntesis, corchetes y llaves.

Ejemplo [A + B] - (3 * (S - 2)) está balanceada.

Nota: No se presentan expresiones de esta forma [A + B + C]. Además, se dispone de un procedimiento que recibe una cola y devuelve dos nuevas colas. En el ejemplo, devolvería una cola con [A + B] y otra con (3 * (S - 2)). Es decir, el procedimiento divide de acuerdo al operador principal.

Referencias bibliográficas

Aho, 1988: Aho, Hopcroft, Ullman "Estructuras de datos y algoritmos", Addison Wesley, 1988.

Garland, 1986: Garland "Introduction to Computer Science with Aplications in Pascal", Addison Wesley, 1986.

Hellman, 1991: Hellman, Veroff "Intermedia Problem solving and Structures", Walls and Mirrows, 1991.

8

Capítulo 8

Análisis de algoritmos: concepto de eficiencia

8

Análisis de algoritmos: concepto de eficiencia

Objetivo

En este capítulo se presenta el concepto de **eficiencia** de un algoritmo, a fin de evaluar diferentes soluciones de un mismo problema.

Se trabaja con dos clases de problemas que constituyen el núcleo de los ejemplos del capítulo: algoritmos de búsqueda y algoritmos de ordenación de datos en vectores.

El análisis teórico de los tiempos de ejecución y/o la memoria ocupada se acompaña con una variada ejemplificación, de modo que el lector pueda analizar algunas soluciones concretas en Pascal.

Asimismo, se presentan casos de soluciones recursivas, mostrando las ventajas de su simplicidad y claridad expuestas en el capítulo 7 y, al mismo tiempo, las limitaciones en cuanto a la eficiencia asociadas con las implementaciones recursivas.

8.1 El concepto de eficiencia

Hasta ahora se han desarrollado algoritmos para expresar la solución de problemas simples, sin analizar profundamente cuán "buena" es la solución propuesta, o si existen alternativas "mejores".

Pensar en la optimización de un algoritmo en algún sentido requiere analizar previamente su eficiencia, es decir, la utilización que se hace desde el algoritmo de los recursos del sistema físico donde se ejecuta (básicamente, tiempo de ejecución en máquina requerido y cantidad de memoria utilizada).

Si bien es cierto que al hablar de eficiencia el mayor énfasis está puesto en el tiempo de ejecución del algoritmo, en un sentido amplio, eficiencia se refiere a la forma de administración de todos los recursos disponibles en el sistema, de los cuales el tiempo de procesamiento es uno de ellos.

Definición

Un algoritmo es eficiente si realiza una administración correcta de los recursos del sistema en el cual se ejecuta.

8.2 Análisis de eficiencia de un algoritmo

En este capítulo se analizarán, básicamente, dos aspectos de la eficiencia de un algoritmo:

- **Tiempo de ejecución**

  Desde este punto de vista se considerarán más eficientes aquellos algoritmos que cumplan con la especificación del problema en el menor tiempo posible. En este caso, el recurso a optimizar es el tiempo de procesamiento. A esta clase pertenecen aquellas aplicaciones con tiempo de respuesta finito como, por ejemplo, la reserva de pasajes, el monitoreo de señales en tiempo real, el control y disparo de alarmas, la transmisión de voz e imágenes en tiempo real, etc.

- **Administración o uso de la memoria**

  Serán eficientes aquellos algoritmos que utilicen las estructuras de datos adecuadas de manera de minimizar la memoria ocupada. Si bien el tiempo de procesamiento no se deja de lado, en este punto se consideran los problemas donde el énfasis está puesto en el volumen de la información a manejar en la memoria (por ejemplo, manejo de bases de datos, procesamiento de imágenes en memoria, reconocimiento de pàtrones, etc).

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

8.3 Análisis de algoritmos según su tiempo de ejecución

8.3.1 ¿Qué medir?

Para poder medir la eficiencia de un algoritmo, desde el punto de vista de su tiempo de ejecución, es fundamental contar con una medida del trabajo que realiza. Esta medida permitirá comparar los algoritmos y seleccionar, de todas las posibles, la mejor implementación.

Básicamente, en los algoritmos se tienen dos operaciones elementales: las comparaciones de valores y las asignaciones. Se comienza con el análisis de cada algoritmo, en base al número de comparaciones y asignaciones involucradas. Naturalmente luego (según el procesador donde el algoritmo se ejecute) se deberá tener en cuenta el tiempo de cada clase de operación.

Ejemplo 8.1

Se considera la tarea de calcular el mínimo de tres números a, b y c y se analizan cuatro formas de resolverlo:

Método 1

```
m := a;
if b<m then m := b;
if c<m then m := c;
```

Método 2

```
if a<=b
  then if a<=c then m := a
               else m := c
  else if b<=c then m := b
               else m := c
```

Método 3

```
if (a<=b) and (a<=c) then  m:= a;
if (b<=a) and (b<=c) then  m:= b;
if (c<=a) and (c<=b) then  m:= c;
```

Método 4

```
if (a<=b) and (a<=c)
  then m:= a
  else if  b<=c then m := b
                else m := c;
```

Analizando la eficiencia relativa de los cuatro métodos puede observarse que:

- El método 1 realiza dos comparaciones y, al menos, una asignación. Puede verse que si las dos comparaciones son verdaderas, se realizan tres asignaciones en lugar de una.
- El método 2 utiliza también dos comparaciones y una sola asignación. Es importante remarcar que el trabajo de un algoritmo se mide por la cantidad de operaciones que realiza y no por la longitud del código.
- El método 3 realiza seis comparaciones y una asignación, ya que aunque las primeras dos comparaciones den como resultado que a es el mínimo, las otras cuatro también se realizan.
- El método 4, si bien requiere una única asignación, puede llegar a hacer tres comparaciones.

Dado que la diferencia entre 2 y 3 o 6 comparaciones puede parecer poco importante, se discute a continuación una modificación del problema:

Hallar el menor (alfabéticamente) de tres *strings* o cadenas en lugar del mínimo de tres números, donde cada uno de los *strings* contenga 200 caracteres. Para comprender aún mejor la situación se supone que el lenguaje no provee un mecanismo para compararlos directamente, sino que debe hacerse letra por letra.

Se puede ver entonces, que hacer dos comparaciones en lugar de tres o seis es importante, pues se convertirán en 400, 600 o 1200, dando una idea del incremento potencial e importante en el tiempo de ejecución.

En este momento, si en lugar de hallar el menor de 3 números o 3 *strings*, se requiere hallar el mínimo de n números o n *strings*, con n grande, es importante ver la calidad de la solución algorítmica propuesta, en el sentido de la posibilidad de generalizar la solución de 3 elementos a n elementos.

Por ejemplo, la generalización del método 3 implica comparar cada número con los restantes, lo cual lleva a realizar n(n-1) comparaciones, mientras que el método 1 solo compara el número buscado una vez con cada uno de los n elementos y haciendo de esta forma (n-1) comparaciones.

Se puede ver que el método 3 requiere de mucho más tiempo que el método 1 ya que realiza n comparaciones por cada elemento.

Como resultado "intuitivo" de este análisis, el método 1 aparece como el más fácil de implementar y generalizar.

Si se identifica la unidad intrínseca de trabajo realizada por cada uno de los cuatro métodos (que en el ejemplo sería la comparación de dos números), es posible determinar cuáles son los

métodos que realizan trabajo superfluo y cuáles los que realizan el trabajo mínimo necesario para llevar a cabo la tarea pedida.

El método 4 no es un buen método en particular, pero sirve para ilustrar un aspecto importante relacionado con el análisis de los algoritmos. Cada uno de los otros métodos presenta un número igual de comparaciones independientemente de los valores de a, b y c. En el método 4, el número de comparaciones puede variar, dependiendo de los datos.

Esta diferencia, se describe diciendo que el método 4 puede realizar dos comparaciones en el **mejor caso,** y tres comparaciones en el **peor caso.**

En general, se tiene interés en el comportamiento del algoritmo en el peor caso o en el caso promedio. En la mayoría de las aplicaciones no es importante cuán rápido puede resolver el problema frente a los datos organizados favorablemente, sino ocurre frente a datos "adversos" (caso peor) o en el caso estadístico de "datos promedio".

Por ejemplo, si se desea encontrar el mínimo de tres números distintos, hay seis casos diferentes correspondientes a los seis órdenes relativos en que pueden aparecer los tres números, en los dos casos donde a es el mínimo, el método 4 realiza dos comparaciones y en los otros cuatro casos, realiza tres comparaciones. De esto se deduce que si todos los casos son igualmente probables, el método 4 realiza 8/3 comparaciones en promedio.

8.3.2 Programas eficientes

A partir de un algoritmo especificado eficientemente, se pueden obtener programas eficientes. Para ello es útil observar algunas pautas simples relacionadas con la eficiencia del código:

- No repetir cálculos innecesarios.
- Por ejemplo, puede escribirse:

  a := 2*x*t;

  y := 1/(a-1) + 1/(a-2) + 1/(a-3) + 1/(a-4)

  en lugar de

  y:=1/(2*x*t -1) + 1/(2*x*t -2) +1/(2*x*t -3) + 1/(2*x*t -4)

  no solo para ahorrar código, sino porque la expresión 2*x*t sólo se calcula una vez.
- El caso anterior se ve agravado si la expresión que se recalcula está dentro de un lazo repetitivo. Suponiendo un lazo que se repite 1000 veces, la ejecución reiterada de 2*x*t significa 4.000 operaciones redundantes.

Ejemplo 8.2

Debe escribirse:

```
t := x*x*x*;
y:=0;
for n:=1 to 2000 do
  y := y + 1/(t-n);
```

en lugar de:

```
y:=0;
for n:=1 to 2000 do
  y := y + 1/(x*x*x-n);
```

ya que se evita calcular x*x*x dos mil veces.

Resulta útil salvar (guardar) los resultados intermedios que se usan en los cálculos posteriores.

```
t := -x*x/2;
p := 1;
y := 1;
for n:= 1 to 2000 do
  begin
    p := p * t;            { p = (-x*x/2)**n }
    y := y + p;            { y = 1 + t + ... + t**n }
  end
```

requiere una única multiplicación dentro del lazo (*loop*).

Los ejemplos anteriores muestran que no necesariamente los códigos más compactos son los más eficientes.

8.3.3 Tiempo de ejecución de un algoritmo

Hay dos formas de estimar el tiempo de ejecución de un algoritmo:

1) Realizar un análisis teórico: se busca obtener una medida del trabajo realizado por el algoritmo a fin de obtener una estimación teórica de su tiempo de ejecución. Básicamente, se calculan el número de comparaciones y de asignaciones que requiere el algoritmo. Los análisis similares realizados sobre diferentes soluciones de un mismo problema permiten estimar cuál es la solución más eficiente.
2) Realizar un análisis empírico: se basa en la aplicación de juegos de datos diferentes a una implementación del algoritmo, de manera de medir sus tiempos de repuesta. La aplicación de los mismos datos a distintas soluciones del mismo problema presupone la obtención de una herramienta de comparación entre ellos. El análisis empírico tiene la ventaja de ser muy fácil de implementar, pero no tiene en cuenta algunos factores como:

- La velocidad de la máquina, esto es, la ejecución del mismo algoritmo en computadoras distintas produce diferentes resultados. De esta forma no es posible tener una medida de referencia.
- Los datos con los que se ejecuta el algoritmo, ya que los datos empleados pueden ser favorables a uno de los algoritmos, pero pueden no representar el caso general, con lo cual las conclusiones de la evaluación pueden ser erróneas.

Por lo tanto, es valioso realizar un análisis teórico que permita estimar el orden del tiempo de respuesta, que sirva como comparación relativa entre las diferentes soluciones algorítmicas, sin depender de los datos de experimentación.

Con ese fin se introduce la siguiente definición: el tiempo de ejecución T(n) de un algoritmo se dice "de orden f(n)" cuando existe una función matemática f(n) que acota a T(n):

Código

```
T(n)=O(f(n)) si existen constantes c y n0
tales que T(n) ≥ c f(n) cuando n≥n0.
```

La definición anterior es una forma de establecer un orden relativo entre las funciones del tiempo de ejecución de los algoritmos T1(n) y T2(n). Según las funciones matemáticas que las acoten (f1(n) y f2(n)) se puede obtener una relación entre T1 y T2.

Por ejemplo, si T1(n) = O(2*n) y T2(n) = O(4*n*n) para n>5, resulta claro que T1 es más eficiente que T2 para n>5.

Otro aspecto a analizar es la velocidad de crecimiento. Por ejemplo, si se compara la función T(n)=1000n con f(n)=n^2, aunque 1000n es mayor que n^2 para valores pequeños de n, n^2 crece más rápido, con lo cual n^2 podría ser eventualmente la función más grande.

En este caso, esto ocurre para n>=1000. La primera definición establece que eventualmente hay algún punto n_0 a partir del cual c*f(n) es al menos tan grande como T(n), e ignorando los factores constantes, f(n) es como mínimo tan grande como T(n).

En el ejemplo, T(n) = 1000n, f(n) = n^2, n_0=1000 y c=1.

También se hubiera podido utilizar n_0=10 y c=100.

Con lo cual es posible afirmar que 1000n = O(n^2) (orden n-cuadrado).

Resumiendo, al decir que T(n)=O(f(n)), se está garantizando que la función T(n) no crece más rápido que f(n), es decir que f(n) es un límite superior para T(n).

Ejemplo 8.3

n^3 crece más rápido que n^2, por lo tanto, se puede afirmar que $n^2 = O(n^3)$.
$f(n) = n^2$ y $g(n) = 2n^2$ crecen a la misma velocidad, por lo tanto ambas expresiones, $f(n)=O(g(n))$ y $g(n)=O(f(n))$ son verdaderas.

Intuitivamente, si g(n) = 2n2, entonces g(n)=O(n4), g(n)=O(n3) y g(n)=O(n2) son todas técnicamente correctas, pero obviamente la última es la mejor respuesta ya que es la cota superior más cercana a g(n).

Volviendo al problema, para calcular el tiempo de ejecución de un algoritmo se analiza el siguiente algoritmo.

Ejemplo 8.4

Se presenta aquí un fragmento de programa que calcula $\Sigma\, i^3$ con i=1..n.

```
function sum( n: integer ): integer;
  var j, sumaParcial: integer;
    begin
{1}   sumaParcial := 0;
{2}   for j := 1 to n do
{3}      sumaParcial := sumaParcial + j*j*j;
{4}   sum := sumaParcial;
    end;
```

El análisis de este programa es simple.
Las declaraciones no consumen tiempo.
Las líneas {1} y {4} implican una unidad de tiempo cada una.
La línea {3} consume 4 unidades de tiempo cada vez (dos multiplicaciones, una suma y una asignación) y es ejecutada n veces dando un total de 4n unidades.
La línea {2} tiene un costo oculto de inicializar j, testear si j≤n e incrementar j. El costo total es 1 por la inicialización, n+1 por todos los testeos y n por todos los incrementos, lo que da 2n+2. (1 + n + 1 + n)
Si se ignora el costo de la invocación y retorno de la función, el total es de 6n+4. Por lo tanto, se dice que esta función es de O(n).

Como puede apreciarse, la aplicación de este tipo de análisis a cada segmento de código es una tarea totalmente impracticable. Sin embargo, hay una cierta cantidad de consideraciones que pueden hacerse a fin de reducir la complejidad de análisis sin afectar la respuesta final.

Por ejemplo, la línea {3} es obviamente de O(1) no importa lo que haya en su interior, es algo que se ejecuta n veces. Lo mismo ocurre con la línea {1}, es insignificante respecto de n.

Existen, por lo tanto, ciertas reglas a tener en cuenta para facilitar esta tarea.

8.3.3.1 Reglas Generales

Regla 1: Para lazos incondicionales.
El tiempo de ejecución de un lazo incondicional es, a lo sumo, el tiempo de ejecución de las sentencias que están dentro del lazo, incluyendo testeos, multiplicada por la cantidad de iteraciones que se realizan.

Regla 2: Para lazos incondicionales anidados.
Se debe realizar el análisis desde adentro hacia afuera. El tiempo total de ejecución de un bloque dentro de lazos anidados es el tiempo de ejecución del bloque multiplicado por el producto de los tamaños de todos los lazos incondicionales.

Regla 3: `If - Then, - Else`
Dado un fragmento de código de la forma

```
if condicion
   then S1
  else  S2
```

el tiempo de ejecución no puede ser superior al tiempo del testeo más el max(t1,t2) donde t1 es el tiempo de ejecución de S1, y t2 el tiempo de ejecución de S2.

Regla 4: Para sentencias consecutivas.
Si un fragmento de código está formado por dos bloques uno con tiempo $T_1(n)$ y otro $T_2(n)$, el tiempo total es el máximo de los dos anteriores, es decir:

Si $T_1(n) = O(f(n))$ y $T_2(n) = O(g(n))$, $T_1(n) + T_2(n) = \text{máx}(\, O(f(n)), O(g(n))\,)$

Analizando un poco más esta última afirmación:

Por definición, $T_1(n) = O(f(n))$ y $T_2(n) = O(g(n))$, de donde existen cuatro constantes c_1, c_2, n_1 y n_2 tales que $T_1(n) \geq c_1 f(n)$ para $n \geq n_1$ y $T_2(n) \geq c_2 g(n)$ para $n \geq n_2$.

Sea $n_0 = \max(n_1, n_2)$, entonces, para $n \geq n_0$, $T_1(n) \geq c_1 f(n)$ y $T_2(n) \geq c_2 g(n)$, de donde $T_1(n) + T_2(n) \geq c_1 f(n) + c_2 g(n)$.

Sea $c_3 = \max(c_1, c_2)$, entonces,

$$\begin{aligned} T_1(n) + T_2(n) &\geq c_3 f(n) + c_3 g(n) \\ &\geq c_3\ (\, f(n) + g(n)\,) \\ &\geq 2c_3\ \max(\, f(n), g(n)\,) \\ &\geq c\ \max(\, f(n), g(n)\,) \qquad \text{para } c=2c_3 \text{ y } n \geq n_0 \end{aligned}$$

A modo de ejemplo, se analiza el siguiente fragmento de código formado por un bloque de O(n) seguido de otro de $O(n^2)$ dando como resultado un fragmento de $O(n^2)$

```
for j := 1 to n do
{ algo para hacer de O(1) };
for j:=1 to n do
   for k:=1 to n do
      { algo para hacer de O(1) };
```

Se puede ver que mediante la utilización de estas reglas el resultado final puede ser sobreestimado, pero nunca subestimado.

8.4 Análisis de algoritmos según su aprovechamiento de memoria

Si bien la evolución y la disminución del costo del hardware parece indicar que la memoria es un recurso cada vez menos crítico, existen muchas aplicaciones en las cuales aún es importante su aprovechamiento.

Desde este punto de vista, la selección de la estructura de datos a utilizar y la forma en que se maneje su contenido determinarán la eficiencia del algoritmo.

8.5 Eficiencia en algoritmos recursivos

La recursividad es una herramienta muy poderosa que puede dar soluciones claras a problemas complejos. Hasta ahora se ha intentado escribir algoritmos recursivos sin analizar la eficiencia de la solución obtenida con el simple objeto de aprender el concepto de recursión.

La recursividad tiene algunas desventajas desde el punto de vista de eficiencia debido a:

- El ***overhead*** (sobrecarga) de tiempo y memoria asociado con la llamada a subprogramas. (Nótese que la recursión por definición involucra un número de autoinvocaciones).
- La ineficiencia inherente de algunos algoritmos recursivos.

Definición

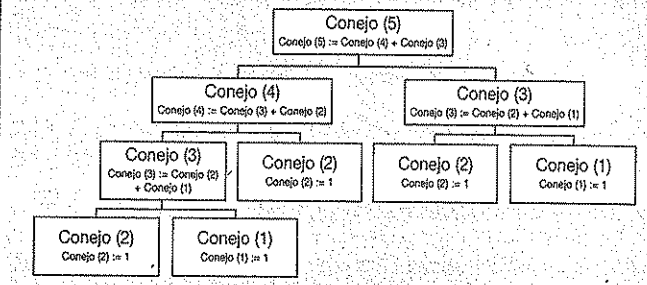
El verdadero uso de la recursividad es como una herramienta para resolver problemas para los cuales no hay una solución iterativa sencilla. En estos casos la recursividad permite obtener una expresión clara del algoritmo en base a la definición del problema original.

Ejemplo 8.5

A continuación se analiza el problema de los conejos (Hellman, 1987) con el fin de mostrar estos conceptos de ineficiencia:

```
Function Conejos(n: word): word;
{ n es un entero positivo }
  begin
    if n <= 2  then Conejos := 1
               else Conejos := Conejos(n-1) + Conejos(n-2);
  end;
```


¿Cómo se calcula, por ejemplo, Conejos(5)?



Como se puede ver, el problema fundamental de esta función es que los mismos valores son calculados una y otra vez. Por ejemplo, Conejos(3) es calculado dos veces y Conejos(2) tres veces. Cuando n es grande, la cantidad de veces que se calcula lo mismo crece considerablemente.

Pese al ejemplo anterior, hay algoritmos recursivos que son muy eficientes y en particular más eficientes que las soluciones no recursivas. En lo que sigue del capítulo se discutirán las soluciones de búsqueda y ordenación de datos, en las que las soluciones recursivas serán muy eficientes. Asimismo, en el tratamiento de árboles y grafos que se verá en el capítulo 9, el lector encontrará que las soluciones de varios algoritmos resultan naturalmente recursivas.

8.6 Estimando el tiempo de ejecución de un algoritmo recursivo

En el caso de los procesos recursivos, no es posible ordenar el trabajo a realizar de modo que se utilicen solo evaluaciones ya realizadas. Lo que se debe hacer ahora es asociar a cada procedimiento recursivo una función de tiempo desconocida T(n), donde n mide el tamaño de los argumentos del procedimiento. Luego se puede obtener una recurrencia para T(n), es decir, una ecuación para T(n) en función de T(k) para varios valores de k.

Si la recursión es utilizada como reemplazo de un lazo iterativo el análisis de tiempo es relativamente fácil.

Ejemplo 8.6

```
Function Factorial( n: integer ): integer;
    begin
{1}   if (n=0) or (n=1)
{2}     then Factorial := 1
{3}     else Factorial := n * Factorial(n-1);
    end;
```

Sea T(n) el tiempo de ejecución de Factorial(n), el tiempo de ejecución para las líneas {1} y {2} es O(1), y para la línea {3} es O(1) + T(n-1). Por lo tanto, para ciertas constantes c y d.

$$T(n) \begin{cases} c + T(n-1) & si\ n > 1 \\ d & si\ n \leq 1 \end{cases} \quad (1)$$

Suponiendo que n>2, se puede desarrollar T(n-1) para obtener
T(n) = 2c + T(n-2) si n>2
Esto es T(n-1) = c + T(n-2), como se puede ver al sustituir n por n-1 en (1). Así pues, es posible reemplazar T(n-1) con c + T(n-2) en la ecuación T(n) = c + T(n-1). Después se puede usar (1) para desarrollar T(n-2), con lo que se obtiene:
T(n) = 3c + T(n-3) si n>3
y así sucesivamente. En general,
T(n) = ic + T(n-i) si n>i
Por último, cuando i=n-1, se obtiene:
T(n) = c (n-1) + T(1) = c (n-1) + d (2).

De (2) se concluye que T(n) es O(n). Es importante observar que en este análisis se ha supuesto que la multiplicación de dos enteros es una operación de tiempo O(1).

Resumiendo, se puede ver que el método general para resolver una ecuación de recurrencia, consiste en reemplazar en forma repetida los términos T(k) del lado derecho de la ecuación, hasta obtener una expresión donde T no aparece.

Por otro lado, cuando la recursión es utilizada por la característica del problema y no como un simple reemplazo a la iteración, el cálculo del tiempo de ejecución puede ser un poco más complejo.

8.7 Algoritmo de búsqueda

8.7.1 Introducción

Los humanos han recolectado información desde mucho antes que las computadoras aparecieran. Esto se ve ejemplificado por los datos de los censos, los registros de libros de las bibliotecas, los precios y las listas de inventarios en un comercio o las fichas de los alumnos egresados de una Universidad; toda esta información y mucha más es almacenada con algún propósito específico.

Dada una colección de elementos, es posible extraer dos tipos de información:

- Información relativa al conjunto de datos de dicha colección
- Información detallada de algún ítem en particular

Volviendo a los ejemplos mencionados en el párrafo anterior, la distribución por edades de la población del país, el número de libros de la biblioteca o el valor del inventario actual de una empresa son ejemplos del primer tipo; en cambio, la edad de una persona o el ISBN de un libro de la biblioteca o el valor de un ítem de un comercio son ejemplos del segundo.

En los casos en que se requiera información particular de un ítem de la colección, este debe ser ubicado y extraído de la misma.

Definición

El proceso de ubicar información particular en una colección de datos es conocido como algoritmo de búsqueda.

Por lo general, se está familiarizado con distintas formas para buscar información. No es lo mismo buscar un número de teléfono en la guía telefónica que buscar una carta en un mazo de cartas y ambos difieren de la manera en que se intenta encontrar la pieza que falta para resolver un rompecabezas.

Como muestran los ejemplos anteriores, la manera de buscar la información depende de la forma en que los datos de la colección se encuentren organizados. Como conclusión, reordenando nuestra información adecuadamente es posible mejorar nuestra capacidad y eficiencia de búsqueda.

Como ejemplo se puede pensar en cómo cambiaría la búsqueda de un número en la guía telefónica si esta no estuviera ordenada alfabéticamente, sino por el orden temporal en que los teléfonos fueron conectados.

En un método de búsqueda típico se cuenta con una lista de ítems que se representarán por a[1], ..., a[n], y es de interés saber cuando un determinado elemento x pertenece a la colección. En caso que x pertenezca a la lista, será importante conocer el valor de j tal que x = a[j], dicho índice será de gran utilidad si se piensan modificar los valores del ítem x en la lista (por ejemplo, modificar el nombre de una persona que ha sido ingresado incorrectamente) o acceder a la información (tal como un número de teléfono) de x en otra lista b[1],..., b[n] asociada a a[1], ..., a[n] por algún criterio.

En todos los casos, los métodos de búsqueda consisten en localizar un dato que tiene una propiedad particular (por ejemplo, una estructura con un componente igual a x) y en caso de encontrarlo, retornar alguna información relacionada con el ítem (por ejemplo, su posición).

A continuación, se desarrolla la búsqueda de datos en colecciones organizadas como vectores. Se utilizarán métodos diferentes teniendo en cuenta si los datos del vector aparecen ordenados, como ocurre con la guía telefónica, o desordenados, como ocurre con las cartas de un mazo de cartas.

8.7.2 Búsqueda Lineal

Cuando se debe buscar un elemento dentro de un vector sin tener información sobre la manera en que este último se encuentra organizado, una forma de proceder es comenzar desde el principio de la estructura, analizando los elementos que contiene uno a uno hasta encontrarlo o hasta llegar al final.

Este tipo de búsqueda es conocida como **búsqueda lineal** ya que procede linealmente (en forma secuencial) a través del vector.

En este punto, donde se están implementando y analizando los diferentes algoritmos de búsqueda son de interés todos los detalles de implementación. Sin embargo en el momento de utilizar uno de estos métodos, estos detalles pierden importancia y solo interesará lo que el algoritmo hace, y no cómo lo hace. Con este criterio, es posible pensar en los algoritmos siguientes, como pertenecientes a bibliotecas intercambiables que se llamarán `busqueda0`, `busqueda1`, etc, dentro de las cuales habrá un método de búsqueda codificado de una manera uniforme como busqueda(a, n, x, j) donde

- a es un vector que contiene n elementos
- x es el ítem a buscar
- j es un índice cuyo valor es calculado por el método de búsqueda utilizado de la siguiente manera: x = a[j] si x pertenece al vector a y j = 0 en caso contrario.

En lugar de decidir *a priori* el tipo de datos sobre los cuales se han de aplicar estos algoritmos, se utilizarán las siguientes declaraciones a fin de dejar este tema para una etapa posterior:

```
const maxlen    = ...          { máxima long.de la lista }
type  tipoElem  = ...          { tipo al cual pertenecen los
                                 elem.del vector }
      indice = 0..maxlen;
      tVector  = array[1..maxlen] of tipoElem;
```

Un primer intento de codificar una búsqueda lineal dará un algoritmo que examina sucesivamente cada elemento de la estructura inicializando ("seteando") el valor del índice j cada vez que encuentre el valor de x en a, o inicializando el valor de j en 0 si nunca lo encontró.

Esto podría representarse a través del siguiente procedimiento:

Ejemplo 8.7

Precondición:
- n <= maxlen

Poscondición:
- j tal que x =a[j] si x es alguno de los elementos de a,
- j = 0 caso contrario


```
{ búsqueda0 : busqueda lineal de x entre a[1], ..., a[n] }
{ ( versión ineficiente )}
procedure buscar(  a: tvector;        { colección de ítems }
                   n: indice;         { cant.de elementos de a }
                   x:  tipoElem;      { ítem a buscar }
                   var j : Indice );  { índice de x }
  var k: indice;
  begin
    j := 0;
    for  k := 1 to n do
      if  x=a[k] then  j := k;
  end;     {fin de la búsqueda}
```

Esta búsqueda es claramente ineficiente. Se puede ver que se recorre el vector completo aunque x sea el primer elemento. Además el vector a está pasado como un parámetro por valor, esto se debe a que el valor del mismo no debe ser modificado por el procedimiento. Esto último es correcto pero tiene el inconveniente que este tipo de pasaje de parámetros implica la construcción de una copia de a, hecho que además de consumir memoria (es importante tener en cuenta que si el tamaño del vector es grande, puede resultar una operación prohibitiva) insume tiempo.

A continuación se presenta una segunda versión del método de búsqueda lineal que pasa el vector como parámetro de entrada, salida para ahorrar tiempo y retorna el valor de j tan pronto como sea posible:

Ejemplo 8.8

Precondición:
- n <= maxlen

Poscondición:
- j tal que x =a[j] si x es alguno de los elementos de a
- j = 0 caso contrario

```
{búsqueda1 : búsqueda lineal de x entre a[1], ..., a[n] }
procedure buscar( var a: tvector;      { colección de ítems }
                      n: indice;       { cant.de elementos de a }
                      x: tipoElem;     { ítem a buscar }
                  var j: indice );     { índice de x }
  var k: word;
  begin
    j := 0;
    k := 0;
    while  (k <= n) and (x <> a[k] ) do
      k := k + 1;
    if  (k>n) then j :=0
              else j := k;
  end;   {fin de la búsqueda}
```

Es común que además de desear saber si un elemento pertenece o no a una colección, sea de interés incorporarlo a la misma en caso de que no pertenezca. Con este objetivo, se desarrolla la tercera alternativa del método de búsqueda lineal:

Ejemplo 8.9

Precondición:
- n <= maxlen

Poscondición:
- j tal que x =a[j]
- n = viejo n + 1 si x no pertenecía al vector
- a[viejo n + 1] = x

```
{búsqueda2 : búsqueda lineal de x entre a[1], ..., a[n]}
{incorpora x al vector si no pertenece}
procedure buscar(var a: tvector;        {colección de ítems}
                 var n: indice;         {cant.de elementos de a}
                     x: tipoElem;       {ítem a buscar}
                 var j: indice );       {índice de x }
  begin
    a[n+1] := x;
    j := 1;
    while  (x <> a[j] ) do
      j := j + 1;
    if  ( j > n) then n := n + 1;
  end;    {fin de la búsqueda}
```

Se puede observar que este último código comienza agregando el elemento buscado al final del vector, de esta forma puede reducir la cantidad de preguntas de cada iteración a la mitad ya que la condición que verificaba si el vector había terminado no necesita ser probada nuevamente, ya que ahora el dato a buscar seguramente existe. Esto redundará en un algoritmo más eficiente que el anterior, dado que la cantidad de consultas realizadas se reduce prácticamente a la mitad, pero tiene como contrapartida una precondición más estricta que antes, dado que ahora n debe ser menor que `maxlen` como una forma de garantizar que el vector tendrá espacio para recibir un componente nuevo.

El ítem ubicado al final del vector en este último algoritmo es conocido como "centinela" ya que es quien evita que el lazo no se repita indefinidamente.

8.7.3 Búsqueda binaria

Cuando los ítems del vector se encuentran ordenados de manera creciente, la búsqueda lineal podría terminar tan pronto como se haya encontrado el elemento buscado o cuando se haya encontrado uno de mayor valor. Sin embargo, existe otra estrategia que puede aplicarse en estos casos.

La **búsqueda binaria** sobre un vector ordenado se basa en la estrategia de "divide y vencerás" y aplica el mismo criterio que el que se utiliza para encontrar el cero de una función continua.

En una búsqueda binaria se compara el ítem buscado con el que se halla en el medio del vector. Si los dos elementos son iguales, la búsqueda ha terminado. Si el elemento buscado es menor que el que se halla en el punto medio del vector, se continúa la búsqueda en la primera mitad del vector, en caso contrario se continúa sobre la segunda mitad. El hecho de que el vector esté ordenado es lo que permite ignorar el 50% de los datos con una única comparación.

Se continúa de esta manera, restringiendo el análisis a porciones cada vez más pequeñas del vector. El proceso termina cuando se encuentra el elemento buscado (cuando coincida con un punto medio) o cuando la porción del vector sea tan pequeña que ya no contenga elementos en cuyo caso se podrá concluir que el elemento no pertenece al vector.

Ejemplo 8.10

Precondición:
- n <= maxlen
- a[1] < a[2]<= ...<= a[n]

Poscondición:
- j tal que x = a[j] si x es alguno de los elementos de a
- j = 0 en caso contrario

```
{ búsqueda3 : búsqueda binaria de x entre a[1], ..., a[n] }
procedure buscar( var a: tvector;       {colección de ítems}
                      n: indice;        {cant.de elementos de a}
                      x: tipoElem;      {ítem a buscar}
                  var j: indice );      {índice de x}
  var pri, ult: indice;                 {límites de la búsqueda}
      medio: indice;                    {punto medio}
  begin
    j := 0;                             {x aun no encontrado}
    pri := 1;                           {búsqueda sobre el}
    ult := n;                           {vector completo.}
    medio := (pri + ult) div 2;
    while (pri<=ult) and (x<>a[medio]) do
      begin
        if x<a[medio] then      ult := medio - 1
                      else      pri := medio + 1;
        medio := (pri + ult) div 2;
      end;
    if pri<=ult then j := medio
                else j := 0;
  end;      { fin de la búsqueda }
```

8.7.4 Análisis de los algoritmos de búsqueda

Intuitivamente se puede observar que la búsqueda binaria en un vector ordenado es más eficiente que una búsqueda lineal. Pero ¿cuánto más eficiente es? Para poder responder a esta pregunta se procede a analizar los algoritmos haciendo un análisis empírico de cada uno de ellos y luego un análisis teórico.

8.7.4.1 Análisis empírico

Para realizar el análisis empírico, se utiliza el siguiente programa que permite calcular el tiempo promedio de cada método de búsqueda.

Código

```
Program TiempoPromedio;
  uses WinCrt, WinProcs, busqueda1;
  var auxi,long: indice;                  { Long.del vector }
      nroElem: indice;                    { índice para buscar }
      vector: tVector;                    { vector con los datos }
      tiempoInicial,                      { comienzo de la búsqueda }
      tiempoFinal: longInt;               { fin de la búsqueda }
      posicion: Indice;                   { lugar donde lo encontró }
  begin
    write('Ingrese la cant.de elementos del vector : ');
    readln(long);
    writeln('vector = ',sizeof( TVector ) );
    { asignamos los valores en el vector en orden creciente }
    for nroElem := 1 to long do
      vector[ nroElem ] := nroElem;
    tiempoInicial := GetCurrentTime;
    auxi := long;
    for nroElem :=1 to auxi do
      begin
        buscar(vector, long, nroElem, posicion);
        if NroElem <> posicion then
        begin
          if posicion=0
            then writeln(nroelem,'no encontrado ')
            else writeln(nroElem,' erróneamente encontrado en', posicion);
        end;
      end;
    tiempoFinal := GetCurrentTime;
    writeln;
    writeln('Tiempo Promedio de búsqueda = ',tiempoInicial, ' ',
            TiempoFinal,' ',(TiempoFinal-TiempoInicial)/long:8:6);
  end.
```


Para analizar los distintos algoritmos de búsqueda es necesario incluir la unidad Pascal correspondiente (desde `busqueda0` hasta `busqueda3`). Este programa mide el tiempo que necesita un algoritmo para encontrar todos los elementos del vector y lo divide por la cantidad de elementos que contiene obteniendo de esta forma el tiempo promedio.

Para hacer esto, se utiliza la función `GetCurrentTime` que retorna la cantidad de milisegundos transcurridos; esta función no pertenece al Pascal estándar, con lo cual el lector deberá adaptarla al compilador que esté utilizando.

La Tabla 8.1 muestra los tiempos promedios (en milisegundos) necesarios para cada algoritmo para hallar un elemento dentro de un vector de 1.000, 3.000 y 5.000 enteros utilizando una AT 686 de 100 Mhz.

Como era de esperar, el tiempo promedio de cada versión de la búsqueda lineal crece linealmente con n, es decir, el tiempo necesario para hallar un ítem en un vector de 5.000 elementos es casi cinco veces lo que tarda para hallarlo en un vector de 1.000.

Por último, el método empírico aplicado en este ejemplo para comparar algoritmos muestra que, el tiempo en el método de búsqueda binaria crece con n más lentamente que en la búsqueda lineal. Esto se justifica mediante el análisis teórico que aparece a continuación. Sin embargo, es importante recordar que para poder aplicar este método, los elementos del vector deben ser ordenados.

Algoritmo	n=1000	n=3000	n=5000
Búsqueda lineal ineficiente (búsqueda0)	1.500	3.843	6.592
Búsqueda lineal (búsqueda1)	900	2.396	3.924
Búsqueda lineal con centinela (búsqueda2)	550	1.830	3.076
Búsqueda binaria (búsqueda3)	0,1	93	110

Tabla 8.1

En realidad, se esperaba una mejora del 50% pero se puede ver que solo llega al 40%. ¿Qué fue lo que ocurrió? El problema está en que el Borland Pascal maneja de una manera más eficiente los lazos tipo `for` que los `while`. De aquí que la ventaja que se pensaba tener reduciendo el número de comparaciones se ve reducida por un incremento del trabajo en cada pasada.

Por último, el método empírico aplicado en este ejemplo para comparar algoritmos muestra que el método de búsqueda binaria tiene una velocidad de crecimiento mucho menor que los lineales. Esto se verá justificado mediante el análisis teórico que aparece a continuación. Sin embargo, es importante recordar que para poder aplicar este método, los elementos del vector deben estar ordenados.

8.7.4.2 Análisis Teórico

Para analizar un algoritmo teóricamente, se debe contar el número de veces que este realiza algún paso esencial. Como los algoritmos de búsqueda deben comparar elementos, se analizará el número de comparaciones expresado como una función de la cantidad de elementos del vector. A continuación se realiza el análisis para el peor caso y el caso promedio de una búsqueda binaria.

Análisis del peor caso en una búsqueda binaria.

Para determinar el peor caso de una búsqueda binaria es necesario contar el número de ítems en el vector que pueden ser encontrados con a lo sumo m comparaciones, es decir, examinando a lo sumo m elementos.

En caso de ser posible realizar una única pregunta solo será posible hallar un elemento, éste será el que se halla en el punto medio del vector. Si a esta pregunta se le pueden agregar dos más, será posible encontrar dos elementos más, el que se encuentra en la posición n/4, siendo n la cantidad de elementos del vector, y el que se halla en la posición 3n/4.

En general, cada prueba adicional permite encontrar el doble de elementos de los obtenidos hasta el paso anterior.

Resumiendo:

con 1 prueba se encuentra	1 ítem	
con 2 pruebas más se encuentran	2 ítems más	∴ se tiene un vector de 3 ítems
con 3 pruebas más se encuentran	4 ítems más	∴ se tiene un vector de 7 ítems
...		
con m pruebas más se encuentran	2m-1 ítems más\tengo un vector de 2m-1 ítems	

De lo anterior se deduce que la búsqueda binaria permite hallar cualquier elemento dentro de un vector de 2^m-1 elementos utilizando a lo sumo m comparaciones.

Dicho de otra forma, si una lista contiene n ítems, la búsqueda binaria utilizará en el peor caso m comparaciones para hallarlo siendo m el menor entero tal que

$2^m - 1 \geq n$

Resolviendo esta desigualdad para m se puede hallar que

$2^m \geq n + 1$ o $m \geq \log_2(n+1)$

donde $\log_2(x)$ es el logaritmo en base 2 de x.

Finalmente, $m = [\log_2(n+1)]$

donde [x] representa el menor entero que es mayor o igual que x.

De este análisis se obtiene que la búsqueda binaria requiere, en el peor caso, un tiempo calculado a través de una función logarítmica contra el tiempo calculado mediante un `modolineal` requerido por la búsqueda lineal, esto es, la búsqueda binaria utiliza un tiempo proporcional al $\log_2(n+1)$, donde n es la cantidad de elementos del vector, mientras que la búsqueda lineal utiliza un tiempo proporcional a n.

Medida de los requerimientos de tiempo de los algoritmos de búsqueda.

La Tabla 8.2 muestra que existen diferencias notables en los recursos requeridos por los algoritmos con tiempos de ejecución que crecen de forma proporcional a n. Los algoritmos que tienen requerimientos de tiempo calculados en forma logarítmica son muchos más eficientes que los algoritmos con requerimientos de tiempo lineal.

Aunque un algoritmo logarítmico realice mayor trabajo que otro lineal por cada paso sobre los datos, siempre será más eficiente para valores grandes de n (esto se puede ver en el algoritmo de búsqueda binario o dicotómico donde por cada punto a analizar realiza más trabajo que el lineal pero dado que se aplica a menor cantidad de datos, resulta ser el más eficiente).

En la Tabla 8.2 se puede ver que las funciones logarítmicas crecen en forma más lenta que la lineales. De la mismá manera, las lineales crecen en forma más lenta que las cuadráticas y éstas que las exponenciales.

Finalmente, los algoritmos exponenciales son totalmente ineficientes aun para valores chicos de n.

Tasa de Crecimiento	N=10	n=100	n=1000
Logarítmica: log2(n)	3,3	6,6	10
Lineal : n	10	100	1.000
Cuadrática : n^2	100	10.000	1.000.000
Exponencial: 2^n	1.024	2^{100}	2^{1000}

Tabla 8.2

Análisis del caso promedio de la búsqueda binaria.

Para poder analizar la cantidad de comparaciones promedio realizadas por la búsqueda binaria es necesario contar el número de comparaciones necesarias para encontrar cada elemento en un vector de longitud n y promediar el resultado.

Se supone por simplicidad que $n = 2^m - 1$. Haciendo una extensión del análisis realizado para el peor caso es posible llegar a la Tabla 8.3.

Nº exacto de Comparaciones	Nº de elementos que pueden ser encontrados	Nº total de pruebas para encontrar todos los ítems
1	1	1 x 1 = 1
2	2	2 x 2 = 4
3	4	3 x 4 = 12
...	...	...
M	2^{m-1}	m x 2^{m-1}

Tabla 8.3

De donde el número total de pruebas requeridas para encontrar todos los ítems del vector es:

$s = (1x1) + (2x2) + (3x4) + ... + m\ 2^{m-1}$

Para expresar esta suma s como una función de n es posible escribir:

$2s = (1x2) + (2x4) + ... + (m-1)\ 2^{m-1} + m\ 2^{m}$

Restando s de 2s se obtiene:

$s = -1 - 2 - 4 - ... - 2^{m-1} + m2^{m}$

Dado que $n = 2^{m} - 1 = 1 + 2 + 4 + ... + 2^{m-1}$

Se llega a que $s = m2^{m} - n$, o lo que es lo mismo $s = (n+1)\log_2(n+1) - n$

Por lo tanto, el número de comparaciones requeridas para encontrar un elemento en un vector de longitud $n = 2^{m} - 1$ es s/n o

$\log_2(n+1) - 1 + (\log_2(n+1) / n)$

el cual es apenas una fracción de una comparación más chica que el número de pruebas requeridas en el peor caso.

Reflexionando sobre el análisis realizado, se observa que es posible utilizar el análisis teórico para extrapolar las estimaciones del comportamiento de un algoritmo de manera de poder aplicarlas a casos no probados. Por ejemplo, la búsqueda binaria en un vector de 10.000 elementos es aproximadamente 44 veces más rápida que utilizando un método de búsqueda lineal. Esto se puede ver en el análisis teórico ya que 5.000 (promedio de comparaciones en la búsqueda lineal) es 44 veces mayor que el $\log_2(10.000-1)$.

A partir de los desarrollos anteriores, se puede concluir que el peor caso es más fácil de calcular que el caso promedio, ya que este último implica tener idea de todas las pruebas a tener en cuenta lo que generalmente concluye en expresiones más complejas.

8.8 Algoritmos de ordenación

En la primera parte de este capítulo se comprobó que la organización de la información permite realizar búsquedas más eficientes. Por ejemplo, el orden alfabético de la guía telefónica permite hallar rápidamente un número de teléfono; en cambio si el orden fuera creciente por número de teléfono, sería fácil saber quién tiene un número dado. En ambos casos, la tarea puede realizarse más eficientemente dado el orden inicial entre los elementos de la estructura.

Definición

El proceso por el cual, un grupo de elementos se puede ordenar se conoce como algoritmo de ordenación.

Los algoritmos de ordenación han sido muy estudiados, particularmente por sus aplicaciones.

A continuación se presentan algunos algoritmos de ordenación en vectores que si bien son muy sencillos de interpretar e implementar son ineficientes. En una etapa posterior se analizarán otros que son más complejos y eficientes.

8.8.1 Consideraciones generales

Los algoritmos de ordenación más sencillos buscan intercambiar los elementos o llevarlos al lugar adecuado de manera de dejar la estructura ordenada.

Como forma de unificar la codificación de los algoritmos de ordenación, se llamará a cada uno `ordenar(a,n)`, donde a es un vector de n elementos.

Los algoritmos de ordenación serán ubicados en distintas bibliotecas `ordenar1`, `ordenar2`, etc., con lo cual, será necesario referenciar la biblioteca correspondiente para poder utilizarlo.

Se supone que cada biblioteca conoce las siguientes declaraciones:

Código

```
const maxlen = ...          { máxima longitud de la lista }
type tipoElem = ...         { tipo al cual pertenecen los elementos
                              del vector }
     indice = 0..maxlen;
     tVector = array[1..maxlen] of tipoElem;
```

8.8.2 Método de selección

Este es el algoritmo de ordenación más sencillo. Para ordenar un vector de n elementos, a[1], ..., a[n], busca el elemento menor según el criterio adoptado y lo ubica al comienzo. Para mantener la dimensión del vector, se intercambia con a[1]. A continuación se busca el segundo elemento menor del vector y se intercambia con a[2]. Continúa así sucesivamente buscando el próximo menor y lo coloca en su lugar, hasta que todos los elementos quedan ordenados.

Para ordenar un vector completo es necesario realizar n-1 pasadas por el vector. En la i-ésima pasada, toma el i-ésimo menor y lo intercambia con a[i]. Luego de n-1 pasadas, el elemento n-1 ha sido ubicado en su lugar y el elemento más grande queda acomodado automáticamente.

La Figura 8.2 presenta el funcionamiento de este método.

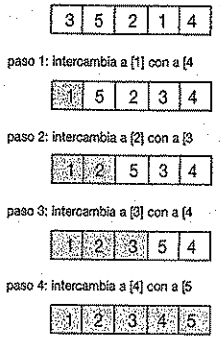


Figura 8.2

El algoritmo podría codificarse de la siguiente forma:

Ejemplo 8.11

```
{Ordenar1: Ordena el vector a por el método de selección }
Procedure Ordenar ( var v: tVector;   { vector a ordenar }
                      n: indice );    { cantidad de elementos }
  var i, j, p: indice;                { índices auxiliares }
      item : tipoElem;                { elemento del vector }
  begin
    for i:=1 to n-1 do
      begin
        { busca el mínimo v[p] entre v[i], ..., v[N] }
        p := i;
        for j := i+1 to n do
          if v[ j ] < v[ p ] then p:=j;
          {intercambia v[i] y v[p] }
          item := v[ p ];
          v[ p ] := v[ i ];
          v[ i ] := item;
          { ahora v[1] <= ... <= v[i] y además, v[i] <= v[j] con i<j<=n }
      end;
  end;
```

Se debe notar que los comentarios, además de ayudar a entender el proceso, pueden utilizarse para indicar las invariantes de los lazos. Esto es lo que ocurre con el último comentario, estas invariantes ayudan en la etapa de corrección de un algoritmo.

8.8.3 Método de intercambio o Burbujeo

Es un algoritmo muy simple de codificar. Procede de una manera similar al de selección, en el sentido de que realiza n-1 pasadas sobre los datos para ordenar un vector de n ítems. Pero a diferencia del anterior, no mueve los elementos grandes distancias, sino que a lo sumo realiza correcciones locales de distancia 1, es decir, compara ítems adyacentes y los intercambia si están desordenados. Al final de la primera pasada, se habrán comparado n-1 pares y el elemento más grande habrá sido arrastrado, como una "burbuja", hacia el final del vector. Al final de la segunda pasada, el segundo máximo habrá sido ubicado en la posición n-1. Esto se ilustra en la Figura 8.3.

El algoritmo en Pascal codificado queda de la siguiente forma:

Ejemplo 8.12

```
{Ordenar2: Ordena el vector a por el método de Intercambios }
Procedure Ordenar ( var v: tVector;   { vector a ordenar }
                      n: indice );    { cantidad de elementos }
  var i, j, p: indice;                { índice auxiliares }
      item   : tipoElem;              { elemento del vector }
  begin
    for i:= n downto 2 do
      begin
      { poner el mayor elemento de v[1],...,v[i] en v[i] }
        for j := 1 to i-1 do
          if v[ j ] > v[ j+1 ] then
            begin                     { intercambiar elementos }
              item := v[ j ];
              v[ j ] := v[ j+1 ];
              v[ j+1 ] := item;
            end;
          {ahora v[i] <= ... <= v[n] y v[j] <= v[i] con 1<=j<i}
      end;
  end;
```

En general, en la i-ésima pasada, se posicionará el máximo entre los i primeros elementos del vector, comparando i-1 pares e intercambiando en cada comparación de ser necesario. Es importante notar que a pesar de no haber cambios, los elementos del vector se siguen comparando hasta terminar, realizando de esta forma un trabajo innecesario.

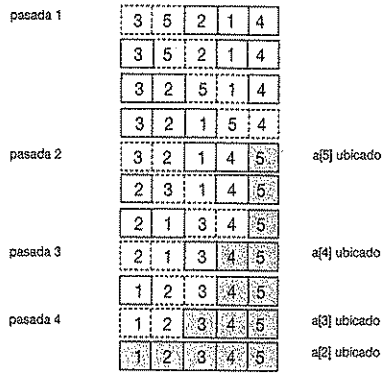


Figura 8.3

Esto puede corregirse fácilmente inspeccionando cuándo ha habido cambios y en caso de no producirse ninguno, se termina el proceso. En el ejemplo 8.13 se muestra el código con esta modificación.

Ejemplo 8.13

```
{Ordenar3: Ordena el vector a por el método de Intercambios }
Procedure Ordenar (var v: tVector;      { vector a ordenar }
                   n: indice );         { cantidad de elementos }
  var i, j, p: indice;                  { indice auxiliares }
      item: tipoElem;                   { elemento del vector }
      huboCambio: boolean;              { indica si hubo cambios }
  begin
    i := n;
    repeat
      { poner el mayor elemento de v[1],...,v[i] en v[i] }
      { setear huboCambio=true si hubo algún cambio }
      huboCambio := false;
      for j := 1 to i-1 do
        if v[ j ] > v[ j1 ] then
        begin
        { intercambiar elementos }
          item := v[ j ];
          v[ j ] := v[ j+1 ];
          v[ j+1 ] := item;
          huboCambio := True;
        end;
      { ahora v[i] <= ... <= v[n] y v[j] <= v[i] con 1<=j<i }
      i := i - 1;
    until( (i < 2) or not HuboCambio );
  end;
```


Pese a estas modificaciones, como se verá más adelante, este método no es precisamente de los más eficiente.

8.8.4 Método de inserción

Este método ordena el vector de entrada insertando cada elemento a[i] entre los i-1 anteriores que ya están ordenados. Para realizar esto comienza a partir del segundo elemento, suponiendo que el primero ya está ordenado. Si los dos primeros elementos están desordenados, los intercambia. Luego, toma el tercer elemento y busca su posición correcta con respecto a los dos primeros. En general, para el elemento i, busca su posición con respecto a los i-1 elementos anteriores y de ser necesario lo inserta adecuadamente.

La figura 8.4 presenta el comportamiento del método.

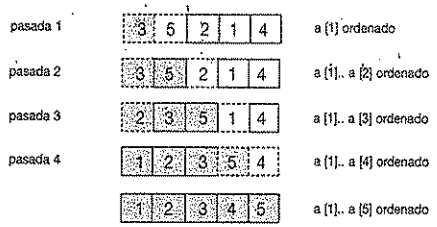


Figura 8.4

El código que aparece en el ejemplo 8.14 muestra la implementación en Pascal del proceso.

Ejemplo 8.14

```
{Ordenar4: Ordena el vector a por el método de Inserción }
Procedure Ordenar (var v: tVector;      { vector a ordenar }
                   n: indice );         { cantidad de elementos }
  var i, j: indice;                     { índices auxiliares }
      item: tipoElem;                   { elemento del vector }
      huboCambio: boolean;              { indica si hubo cambios}
  begin
    for i:=2 to n do
      begin
        item := v[i];
        j := i-1;      { apunta al primer elemento a comparar }
        while (j>0) and ( v[ j ] > item ) do
          begin
            v[ j+1 ]:= v[ j ];      { se corre 1 lugar a la derecha }
            j := j - 1;
          end;
        v[ j+1 ] := item;           { inserta el item }
      end;
  end;
```

8.8.5 Análisis de los métodos de ordenación elementales

Los algoritmos anteriores ordenan en forma creciente. El mejor caso es encontrar el vector ordenado y el peor caso es que el vector esté ordenado en orden inverso.

Con el fin de medir el tiempo promedio de cada uno de los algoritmos anteriores y dado que no es posible medir el tiempo que cada uno emplea para ordenar vectores en todos los posibles órdenes iniciales, se medirá el tiempo que tarda en ordenar un vector cuyos valores iniciales fueron asignados al azar.

Para generar un vector con valores al azar se usará la función `Random(n)` que genera un número al azar entre 0 y n-1, de la siguiente forma:

Ejemplo 8.15

```
Procedure  AlAzar( var v: tvector; n: indice );
{asigna a v valores enteros positivas de manera aleatoria o random}
  var i,j: indice;
      item: tipoElem;
  begin
    randomize;    {para comenzar la generación de números al azar}
    for i:=1 to n do
      v[i] := Random( n ) + 1;
  end;
```

Utilizando todo lo visto hasta ahora, el programa que mide el tiempo de ejecución de un método de ordenación quedaría así:

Ejemplo 8.16

```
Program TiempoOrdenacion;
  Uses WinCrt, WinProcs, Ordnear1;

  Procedure AlAzar( var v: tvector; n: indice );
  {asigna a V valores enteros positivos de manera aleatoria o random}
    var j: indice;
    begin
      randomize; {para comenzar la generación de números al azar}
      for j:=1 to n do
        v[j] := Random( n ) + 1;
    end;
```



```
Procedure Check( var v: tvector; n: indice );
    var tiempoInicial,          { Comienza la ordenación }
        tiempoFinal: longInt;   { Fin de la ordenación }
        j: indice;

    begin
      tiempoInicial := GetCurrentTime;
      ordenar( v, n);
      tiempoFinal := GetCurrentTime;
      writeln('Tiempo =', tiempoFinal - tiempoInicial);
      j := 2;
      while (j<=n) and (v[j-1]<=v[j]) do
        j:=j+1;
      if j>n  then writeln('Vector Ordenado')
              else writeln('Vector DESORDENADO !!!!!');
    end;

  var j: integer;
      v: tvector;
      n: indice;
  begin
    { Primer prueba: Vector pre-ordenado  }
    for j:=1 to n do
        v[j] := j;
    writeln;
    writeln('Ordenando un vector pre-ordenado');
    Check( v, n);
    { Segunda prueba: Vector en orden inverso  }
    for j:=1 to n do
        v[j] := n-j+1;
    writeln;
    writeln('Ordenando un vector en orden inverso ');
    Check( v, n);
    { Tercer prueba: Vector en orden random  }
    AlAzar( v, n );
    writeln;
    writeln('Ordenando un vector desordenado ');
    Check( v, n);
  end;
```

Si se utiliza este programa para medir tiempos para los cuatro algoritmos anteriores (cambiando la biblioteca cada vez para poder referenciar el método correspondiente), se obtendrán resultados comparativos notables.

A continuación se muestran los resultados de ejecutar el programa para vectores de 500 y 1.000 elementos con diferentes órdenes iniciales. La Tabla 8.4 presenta el tiempo en milisegundos para ordenar un vector de 500 elementos para los tres algoritmos citados. La Tabla 8.5 presenta el tiempo en milisegundos para ordenar un vector de 1.000 elementos con los tres métodos descritos.

Método	Ordenado	Orden Inverso	Orden Random
Selección	55	55	55
Burbuja	55	110	70
Burbuja2	0	120	62
Inserción	0	70	32

Tabla 8.4

Método	Ordenado	Orden Inverso	Orden Random
Selección	109	110	110
Burbuja	110	274	219
Burbuja2	0	275	220
Inserción	0	164	110

Tabla 8.5

En los cuadros anteriores se puede ver que el método de selección y el de inserción compiten por el primer lugar. El método de la burbuja con centinela, `burbuja2`, es bueno cuando el vector está ordenado pero, de no ser así, es tan ineficiente (en general, un poco más ineficiente) como el método de intercambio común.

8.8.6 Análisis del peor caso de los algoritmos de ordenación

Para poder medir los diferentes algoritmos de ordenación es necesario calcular la cantidad de trabajo que realiza cada uno. Esto implica contar la cantidad de comparaciones (C) y la cantidad de intercambios (I) que realizan. Se denomina intercambio al movimiento de un ítem dentro del vector. Obviamente, mientras más grande sea la cantidad de comparaciones e intercambios que realice un método, más tardará.

8.8.6.1 Método de selección

Durante la primera pasada, cuando selecciona el menor elemento dentro del vector, realiza n-1 comparaciones. Sobre la segunda pasada, realiza n-2 comparaciones para hallar el segundo mínimo. En general, en la i-ésima pasada realiza n-i comparaciones para hallar el i-ésimo mínimo. Por lo tanto, el total de comparaciones es:

$$C_{seleccion} := (n-1) + (n-2) + \ldots + 2 + 1$$

para poder hallar una expresión para la cantidad de comparaciones se realiza el siguiente cálculo:

$$C_{seleccion} := (n-1) + (n-2) + \ldots + 2 + 1$$

$$+ \quad C_{seleccion} := 1 + 2 + \ldots + (n-2) + (n-1)$$

$$2C_{seleccion} := n + n + \ldots + n + n$$

de donde:

$C_{seleccion} := n\,(n-1)\,/2$ que como vemos es de orden n^2.

La cantidad de intercambios del método de selección es $I_{seleccion} := 3(n-1)$, ya que se necesitan n-1 pasadas para ordenar el vector y por cada una de ellas se intercambian dos elementos del vector, lo que implica 3 movimientos.

8.8.6.2 Método de la burbuja

Si se busca medir la cantidad de comparaciones, para la primera pasada realiza n-1 comparaciones, para la segunda, n-2, en general, para la i-ésima pasada realiza n-i comparaciones. Haciendo un razonamiento similar al realizado para el método de selección es posible afirmar que:

$C_{burbuja} := n\,(n-1)/2$

y dado que en cada comparación se intercambian dos elementos del vector, se realizan tres movimientos por cada vez (en el peor caso), de donde:

$I_{burbuja} := 3\,n\,(n-1)\,/2$

La cantidad de intercambios realizados por el método de la burbuja explica la demora que se refleja en el análisis empírico realizado anteriormente.

8.8.6.3 Método de inserción

Este método realiza, en el mejor caso una sola comparación (solo con el elemento anterior y ve que está ordenado) y en el peor caso, tantas comparaciones como elementos anteriores tenga (el peor caso es que lo tenga que insertar al comienzo del vector), por lo tanto:

$n-1 <= C_{insercion} <= n\,(n-1)\,/2$

En cuanto a la cantidad de intercambios, en el mejor caso, el elemento analizado debe quedar donde está, esto implica dos movimientos (ver implementación). En cambio, en el peor caso debe realizar además de estos dos movimientos, un movimiento más por cada elemento que

deba desplazar hacia la derecha para generar el lugar en el vector donde insertar y como el peor caso es la inserción en el primer lugar, este número de movimientos será igual a la cantidad de comparaciones realizadas en el peor caso. Por lo tanto:

$$2(n-1) <= I_{insercion} <= 2(n-1) + n\,(n-1)\,/2 = (n+4)(n-1)/2$$

Queda como ejercicio para el lector comparar estos resultados teóricos con los valores obtenidos de manera empírica y obtener las conclusiones.

8.9 Ordenación por índices

En algunas ocasiones existe la necesidad de ordenar información que está compuesta por datos heterogéneos, algunos de los cuales pueden servir como criterio para ordenar los elementos y otros solo llevan información adicional.

Esto se puede ver en el siguiente ejemplo:

Ejemplo 8.17

Se dispone de cierta información sobre los alumnos de un curso y se desea obtener un listado ordenado por nombre y apellido o por el nombre de la ciudad de origen del alumno.

La información que se utiliza para determinar el orden buscado se conoce como clave o base de comparación para el algoritmo de ordenación.

Como forma de generalizar el campo a utilizar se puede escribir la siguiente definición de registro:

```
Type tipoElem = record
                clave : tipoClave;
                ...
     end;
```

de donde será necesario modificar el algoritmo de ordenación para poder utilizar la clave.

Esto se logra simplemente cambiando:

```
if  v[j] < v[p]  then...
por if v[j].clave < v[p].clave then...
```

y dejando el resto del algoritmo como está.

Si la información tuviera más de una clave por la que interesara ordenar, no será sencillo expresar la representación en Pascal. Un ejemplo puede ser el siguiente:

```
type tipoElem = record
                clave1: tipoClave;
                clave2: tipoClave;
                ....
     end;
```

esta complementación no parece ser una forma muy conveniente. Otra representación que puede resultar más conveniente consiste en estructurar los elementos como vectores en lugar de registros, de esta forma podría agregarse un parámetro al módulo a fin de indicar qué componente del vector se utilizará como índice. Sin embargo, mediante esta representación se pierde la posibilidad de disponer de tipos distintos dentro de un mismo ítem, situación que no ocurría utilizando el tipo de datos registro.

Si la información a ordenar estuviera organizada en forma de matriz, en lugar de un vector, donde cada columna contiene la misma clase de información y cada fila representa los datos de un mismo objeto, por ejemplo, los datos de una persona, ordenarla por algún criterio implicará mover una gran cantidad de información, ya que independientemente del criterio elegido, cada fila deberá mantenerse unida. Por ejemplo, si se ordena la matriz por nombre, no bastará con intercambiar el contenido de dos celdas de la matriz, sino que deberán intercambiarse dos filas completas.

Ejemplo 8.18

La Figura 8.5 representa una matriz con datos de 6 empleados de diferentes sucursales de una misma empresa. Las columnas indican: código de sucursal, apellido, nombre y dirección, respectivamente.

Columnas

Filas

A	Gutierrez	Luciana	Benito de Miguel 34
B	Martire	Ruben	Saenz Peña 234
C	Rivera	Simon	Garay 348
D	Alvarez	Pedro	Cordoba 1200
E	Alvarez	Cristobal	Santa Fe 500
F	Marcico	Natalia	J.B. Justo 400

Figura 8.5

y se busca obtener como resultado la matriz de la figura 8.6

Columnas

Filas

E	Alvarez	Cristobal	Santa Fe 500
D	Alvarez	Pedro	Cordoba 1200
A	Gutierrez	Luciana	Benito de Miguel 34
F	Marcico	Natalia	J.B. Justo 400
B	Martire	Ruben	Saenz Peña 234
C	Rivera	Simon	Garay 348

Figura 8.6

donde los empleados aparecen ordenados alfabéticamente por apellido (contenido de la columna 2).

El procedimiento `OrdenarFilas` de la biblioteca `Xordenar0` es una modificación del método de selección. Su función es ordenar las filas de la matriz de entrada según los valores de la columna *k*. Para ello, utiliza las siguientes declaraciones:

Ejemplo 8.19

Precondición:

- 1 <= k <= m

Poscondición:

- a ordenado según los valores de la columna k, es decir, a[1,k]<=a[2,k]<= ... <= a[n,k]

Se utilizarán las siguientes declaraciones:

```
const maxFilas = ... {máxima cantidad de filas}
      maxColum = ... {máxima cantidad de columnas}
type tipoElem = ...  {tipo al que pertenece los elementos de la matriz}
     indiceFila = 0..maxFilas;
     indiceCol = 0..maxColum;
     matriz  = array[1..maxFilas, 1..maxColum] of tipoElem;

{Xordenar0 : ordena las filas de la matriz a(nxm) usando como clave
             el contenido de la columna k }

Procedure OrdenarFilas(var a: matriz;     {matriz de datos }
                           n: indiceFila; {cantidad de filas de a}
                           m: indiceCol;  {cantidad de columnas de a}
                           k: indice;     {nro. de columna con la clave}
  var  i,j,p: indiceFila;
```



```
       c: indiceCol;
       item: tipoElem;

  begin
    for i:=1 to n-1 do
      begin
        { busca el valor de p que contiene el mínimo entre a[i,k], ..., a[n,k] }
        p:=i;
        for j:=i+1 to n do
          if a[j,k]<a[p,k] then p:=j;
        {Intercambia la fila i con la fila p }
        for c:= 1 to m do
          begin
            item := a[i,c];
            a[i,c] := a[p,c];
            a[p,c] := item;
          end;
          { ahora a[1,k] <= ... <= a[i,k] y a[i,k] <= a[j,k] con i<j<=n }
      end;
  end;
```

Como se puede ver en el código anterior, cada vez que se necesita intercambiar dos valores en la columna k debe intercambiarse el valor de dos filas completas de la matriz.

Esto mismo puede mejorarse si en lugar de intercambiar físicamente las filas de la matriz se utiliza una estructura externa de tipo vector que indique la forma en que las filas deben ser leídas para "verse" ordenadas por el criterio indicado. Es decir, que la matriz permanecería inalterada mientras que las distintas modificaciones se harían sobre el arreglo. De esta forma, los intercambios vuelven a ser intercambios de elementos y no de filas.

A continuación se incorpora la declaración de la estructura que permite manejar este índice.

```
type TipoIndice = array[1..MaxFilas] of 1..Maxfilas;
```

El algoritmo se puede modificar como sigue.

Ejemplo 8.20

Precondición:

- 1<=k<=m

Poscondición:

- x ordenado según los valores de la columna k, es decir,

  a[x[1],k]<=a[x[2],k]<= ... <= a[x[n],k]

```
{XOrdenar1 : ordena el vector de índices x usando como clave el
             contenido de la columna k }

Procedure  OrdenarFilas( var a: matriz;     {matriz de datos}
                         var x: tipoIndice {índice de la matriz}
                         n: indiceFila;    {cantidad. de filas de a}
                         k: indice;   {nro. de columna con la clave}
  var i,j,p: indiceFila;
  begin
    for i:=1 to n-1 do
      begin
        {busca el mínimo a[x[p],k] entre a[x[i],k], ..., a[x[n],k]}
        p:=i;
        for j:=i+1 to n do
          if a[x[j],k]<a[x[p],k] then p:=j;
        {Intercambia x[i] con x[p] }
        j := x[i];
        x[i] := x[p];
        x[p] := j;
        {ahora a[x[1],k] <= ... <= a[x[i],k] y a[x[i],k] <= a[x[j],k] con }
        {i<j<=n}
      end;
  end;
```

Las Figuras 8.7 a 8.12 presentan las modificaciones que se realizan sobre el vector de índice a medida que se ejecuta el algoritmo utilizando como clave la comuna 2.

1	A	Gutiérrez	Luciana	Benito de Miguel 34
2	B	Martire	Rubén	Saenz Peña 234
3	C	Ribera	Simón	Garay 348
4	D	Alvarez	Pedro	Córdoba 1200
5	E	Alvarez	Cristóbal	Santa Fe 500
6	F	Marcico	Natalia	J.B. Justo 400

Figura 8.7: Estado inicial del índice x y el arreglo

5	A	Gutiérrez	Luciana	Benito de Miguel 34
2	B	Martire	Rubén	Saenz Peña 234
3	C	Ribera	Simón	Garay 348
4	D	Alvarez	Pedro	Córdoba 1200
1	E	Alvarez	Cristóbal	Santa Fe 500
6	F	Marcico	Natalia	J.B. Justo 400

Figura 8.8: Estado del índice y del arreglo luego de la primer pasada

5	A	Gutiérrez	Luciana	Benito de Miguel 34
4	B	Martire	Rubén	Saenz Peña 234
3	C	Ribera	Simón	Garay 348
2	D	Alvarez	Pedro	Córdoba 1200
1	E	Alvarez	Cristóbal	Santa Fe 500
6	F	Marcico	Natalia	J.B. Justo 400

Figura 8.9: Estado del índice y del arreglo luego de la segunda pasada

5	A	Gutiérrez	Luciana	Benito de Miguel 34
4	B	Martire	Rubén	Saenz Peña 234
1	C	Ribera	Simón	Garay 348
2	D	Alvarez	Pedro	Córdoba 1200
3	E	Alvarez	Cristóbal	Santa Fe 500
6	F	Marcico	Natalia	J.B. Justo 400

Figura 8.10: Estado del índice y del arreglo luego de la tercer pasada

5	A	Gutiérrez	Luciana	Benito de Miguel 34
4	B	Martire	Rubén	Saenz Peña 234
1	C	Ribera	Simón	Garay 348
6	D	Alvarez	Pedro	Córdoba 1200
3	E	Alvarez	Cristóbal	Santa Fe 500
2	F	Marcico	Natalia	J.B. Justo 400

Figura 8.11: Estado del índice y del arreglo luego de la cuarta pasada

5	A	Gutiérrez	Luciana	Benito de Miguel 34
4	B	Martire	Rubén	Saenz Peña 234
1	C	Ribera	Simón	Garay 348
6	D	Alvarez	Pedro	Córdoba 1200
2	E	Alvarez	Cristóbal	Santa Fe 500
3	F	Marcico	Natalia	J.B. Justo 400

Figura 8.12: Estado del índice y del arreglo luego de la quinta pasada

El valor final del vector indica que el empleado con el primer apellido (según orden alfabético) es el 5, y luego 4, 1, 6, 2 y 3. La matriz permanece sin cambios, es decir, si no se utiliza el vector para procesar la matriz, el método de ordenación de `Xordena1` pasará inadvertido.

Queda como ejercicio para el lector el análisis empírico de la eficiencia de éstos dos últimos métodos.

8.10 Métodos de ordenación eficientes

Los algoritmos de ordenación descritos anteriormente requieren un tiempo proporcional a n^2. Es posible resolver los problemas de una manera más eficiente, utilizando un algoritmo cuyo tiempo promedio de ejecución sea proporcional a $n \log_2(n)$.

Los métodos que aparecen a continuación son un ejemplo del uso de la recursividad.

8.10.1 *Sorting by Merging*

Con la idea de aplicar la técnica de "divide y vencerás", es posible dividir el problema de ordenar un vector de n elementos en dos subproblemas:

- Ubicar el elemento menor en la primera posición
- Ordenar el resto de los elementos (a partir del segundo en adelante)

Como se puede observar, esta división no representa dos tareas de igual complejidad. La intuición indica que si el trabajo estuviese dividido de manera más equitativa sería posible obtener un algoritmo más eficiente. La idea es pensar en dos tareas que resueltas por dos personas distintas puedan resolver el problema en menos tiempo.

¿Cuál sería la nueva división del problema de ordenación?

- Dividir el vector en dos partes iguales de n/2 elementos
- Ordenar cada subvector
- Hacer un *merge* de los dos subvectores para obtener el resultado final

Dado que el proceso de *merge* no insume mucho tiempo, parece razonable pensar en que se podría obtener una reducción del mismo.

En este caso es necesario modificar la interfaz de los algoritmos de ordenación, ya que ahora deben ordenar una porción del vector, por lo tanto, deberán recibir un índice inicial y final que indican el subrango sobre el cual debe trabajar.

```
procedure Ssort( Var a : Vector; inicio, fin : Indice );
```


El algoritmo de ordenación puede escribirse de la siguiente forma:

Ejemplo 8.21

```
procedure  msort1( var a: vector; inicio, fin: indice );
  var  medio: indice;
  begin
     medio := (inicio+fin) div 2;
     sort( a, inicio, medio );
     sort( a, medio + 1, fin);
     merge( a, inicio, medio, fin );
  end;
```

Utilizando esta idea, la cantidad de comparaciones se reduce a la mitad. La Figura 8.13 presenta el análisis de la solución.

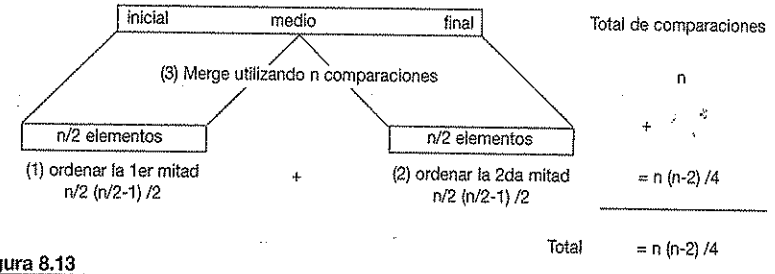


Figura 8.13

Ejemplo 8.22

```
Procedure Merge( var a: tVector; inicio, medio, fin : Indice );
  var v: tvector;      {vector auxiliar para ir armando la mezcla}
      j,k: indice;     {indices para recorrer los subvectores}
  begin
    j:= inicio;
    k := medio + 1;    {medio señala el final del primer subvector}
    i:= 0;             {vector de salida actualmente vacío }
    while (j<=medio) and (k<=fin) do {mientras haya elementos en ambos}
      begin                          {subvectores}
        i:=i+1;
        { copia a la salida el elemento más chico }
        if a[j] < a[k] then
        begin
          v[i] := a[j];
          j := j + 1;
        end
```

```
        else  begin
          v[i] := a[k];
          k := k + 1;
        end
      end;
      {al menos uno de los subvectores ha terminado, hay que copiar}
      {lo que queda del otro.}
      while (j<=medio) do {mientras haya elementos en el 1er.subvector}
        begin
          i:=i+1;
          v[i] := a[j];
          j := j + 1;
        end;
        while (k<=fin) do {mientras haya elementos en el 2do.subvector}
          begin
            i:=i+1;
            v[i] := a[k];
            k := k + 1;
          end;
          {falta asignar v al vector de salida a}
          a := v;
  end;
```

A partir del algoritmo anterior se puede ver que el tiempo empleado para unir adecuadamente los dos subvectores es O(n).

Siguiendo con el razonamiento de "divide y vencerás", es posible volver a aplicar el método dividiendo el vector en cuatro partes en lugar de dos. Para implementarlo, se puede ver el siguiente código:

Ejemplo 8.23

```
procedure  msort2( var a: vector; inicio, fin: indice );
  var  medio: indice;
  begin
    medio := (inicio+fin) div 2;
    msort1( a, inicio, medio );
    msort1( a, medio + 1, fin);
    merge( a, inicio, medio, fin );
  end;
```

Si se calcula la cantidad de comparaciones necesarias para `msort2` se puede ver que cada subvector insume, para ordenarse, n(n-4)/8 comparaciones y, en cuanto a los *merges*, se necesitan n/2 comparaciones para unir dos vectores de longitud n/4, de donde se necesitan n comparaciones para formar dos vectores de n/2 elementos a partir de 4 vectores de longitud n/4 y otras n

comparaciones para formar un vector de n elementos a partir de dos de longitud n/2. Esto da un total de n(n+12)/8 comparaciones, es decir, la mitad de las realizadas por `msort1`.

Se puede ver que el algoritmo será cada vez más eficiente a medida que se vaya dividiendo el vector en mas partes. Con esta idea se escribe el siguiente código:

Ejemplo 8.24

```
procedure  msort( var a: tvector; ini, fin: indice );
  var medio: indice;
  begin
    if ini < fin then
    begin
      medio := (ini + fin) div  2;
{1}   msort( a, ini, medio );
{2}   msort( a, medio +1, fin);
{3}   merge( a, ini, medio, fin);
    end;
  end;
```

A continuación se analiza su funcionamiento.

Ejemplo 8.25

Se desea ordenar el siguiente vector

a = [8 4 1 3]

habrá una invocación no recursiva al procedimiento anterior de la siguiente forma:

`msort(a, 1, 4);`

gráficamente se puede ver la ejecución en la Figura 8.14.

¿Qué tan eficiente es el algoritmo `msort`? Este algoritmo trabaja dividiendo el vector por la mitad en forma iterativa hasta que cada subvector contiene un único ítem. Llegado a este punto no se necesitan más comparaciones para ordenar los subvectores, sino que es el procedimiento *merge* el que los acomoda para que queden ordenados.

Por lo tanto, se necesitan $[\log_2(n)]$ procesos de *merge* para ordenar el vector: el primero ordenando un subvector de 2 elementos, el segundo de 4 elementos y así sucesivamente.

Cada uno de estos *merges* requieren, al igual que en `msort1`, n comparaciones como máximo, de donde se tiene que `msort` requiere $n[\log_2(n)]$ comparaciones.

La ventaja de `msort` respecto del `ssort` es notable lo que se puede observar al calcular los valores de n^2 y $n[\log_2(n)]$ para varios valores de n. (Tabla 8.6)

Long. del Vector n	Ordenación lenta N^2	Ordenación rápida $n\ [\log_2(n)]$
10	100	14
100	10.000	700
1.000	1.000.000	10.000
10.000	100.000.000	140.000
100.000	10.000.000.000	1.700.000

Tabla 8.6

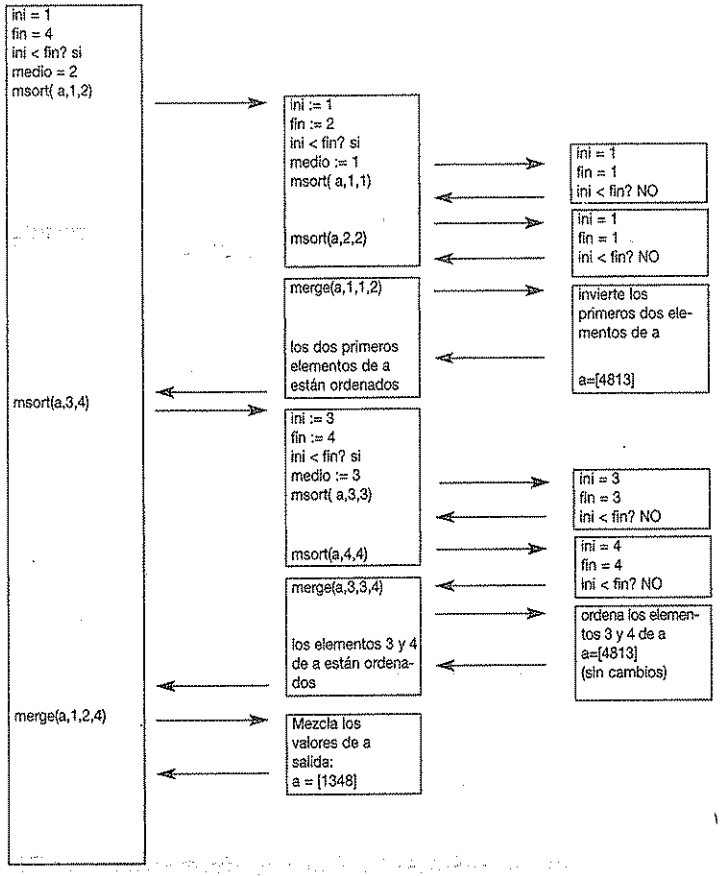


Figura 8.14

Se puede ver que el método de selección requiere 10 billones de comparaciones para ordenar un vector de 100.000 elementos mientras que el método de *sort by merge* recursivo necesita menos de 2 millones de comparaciones.

Como conclusión se puede decir que este último método de ordenación es mucho más eficiente que los métodos anteriores no recursivos.

Otro aspecto muy importante que presenta es que cada uno de los procesos de ordenación parcial podría ejecutarse al mismo tiempo en diferentes procesadores. Este procedimiento de ejecución en paralelo incrementa notoriamente la eficiencia en cuanto al tiempo.

Sin embargo, este beneficio tiene un costo. El algoritmo de *merge* invocado por `msort` necesita memoria auxiliar para mezclar los datos ordenadamente. Si el vector es una estructura muy grande puede resultar prohibitivo utilizar otra estructura de igual tamaño para realizar la mezcla.

Por esto, se presenta a continuación un algoritmo recursivo que es del mismo orden que `msort` pero que no necesita memoria auxiliar

Ordenación rápida (*Quicksort*)

Este algoritmo, al igual que el *sort by merge* recursivo, requiere un número de comparaciones proporcional a $n \log_2(n)$, en promedio, para ordenar un vector de n elementos. Sin embargo, es mejor que `msort`, dado que no necesita un almacenamiento auxiliar.

El `QuickSort` es un método recursivo al igual que el *sort by merge* pero no utiliza el *merge*, sino que trabaja ordenando dos subvectores de la mitad de tamaño que el original donde todos los elementos que se encuentran en la primera mitad son menores que los de la segunda mitad, con lo cual, luego de ordenar ambas partes, el vector completo queda ordenado directamente (el *merge* no es necesario).

El algoritmo se puede expresar de la siguiente forma:

Ejemplo 8.26

```
{ QuickSort }
if lista no vacía then
   begin
     dividir la lista en dos de manera que los elementos de la primer
      mitad sean menores que los de la segunda.
     ordenar la primer mitad.
     ordenar la segunda mitad.
   end
```

Ejemplo 8.27

Supóngase tener el siguiente vector de entrada:

25 36 22 41 57 48 52

se puede ver que los primeros 3 elementos son menores que 41 y que los últimos 3 son mayores que 41, con lo cual, ordenando los 3 primeros elementos entre sí y los 3 últimos entre sí, se tendrá el vector completo ordenado. En esta situación el *merge* no es necesario.

Sin embargo, no es común trabajar con vectores que tengan esta característica, de donde, será necesario contar con un algoritmo que permita dividir el vector original en dos partes, donde la primera mitad sea menor que la segunda mitad.

Se intentará resolver este problema de la siguiente forma, donde se supone el siguiente vector de seis elementos:

17 36 22 41 12 24

se intentará reubicar el 24 de manera que todos los que se encuentren a su izquierda sean menores que él, y los de la derecha sean mayores.

Provisoriamente, se dejará el 24 en un lugar auxiliar y se ubicarán dos marcas, der e izq en los extremos del vector

17	36	22	41	12	**	24
izq					der	

se comienza corriendo izq hacia la derecha hasta encontrar un elemento que sea mayor que 24

17	36	22	41	12	**	24
	izq				der	

y se pone el elemento hallado en el hueco dejado por el 24

17	**	22	41	12	36	24
	izq				der	

Ahora, se desplaza la marca der hacia la izquierda buscando un elemento que sea menor que 24

17	**	22	41	12	36	24
	izq			der		

y se lo ubica en el hueco que hay en izq

17	12	22	41	**	36	24
	izq			der		

Se repite el proceso corriendo izq hacia la derecha buscando un elemento mayor que 24

17	12	22	41	**	36	24
			izq	der		

y se lo ubica en la posición indicada por der.

17	12	22	**	41	36	24
			izq	der		

Se desplaza der hacia la izquierda buscando un elemento menor que 24 o hasta alcanzar a izq.

17	12	22	**	41	36	24
			izq			
			der			

cuando las marcas se unen, indican el lugar donde ubicar el 24.

17	12	22	24	41	36
			izq		
			der		

Se ha obtenido de esta forma un vector donde los valores que están a la izquierda del 24 son menores que él y los que están a la derecha son mayores.

Este proceso puede implementarse en Pascal de la siguiente forma:

Código

```
procedure Dividir( var a: tvector;          {vector a dividir}
                       pri, ult: indice;    {límites del subvector}
                   var medio: indice;       {punto de división}
  {reordena a[pri], .., a[ult] y setea el valor de medio de manera que}
  {a[ini], .., a[medio-1] <= a[medio] <= a[medio+1], ..., a[ult]}
  var izq, der: indice;
      x: tipoElem;
  begin
    {se elige un x en el vector y se marca izq y der de modo que}
    {(1) a[i] <= x        si ini<=i<izq       }
    {(2) x <= a[j]        si der < j <= ult   }
    {(3) existe un lugar para x en a[der]     }
    x := a[ult];
    izq := ini;
    der := ult;
    { se mueve izq y der preservando de (1) a (3)  }
    while izq < der do
      begin
      { se  mueve la marca izq hacia la derecha }
        while (izq<der) and (a[izq]<=x) do
          izq := izq + 1;
        if izq < der then
        begin
          a[der] := a[izq];
          der := der - 1;
        end;
      { se mueve la marca der hacia la izquierda }
        while (izq<der) and (x <= a[der]) do
          der := der - 1;
```

```
        if izq < der then
        begin
          a[izq] := a[der];
          izq := izq + 1;
        end;
      end;
    medio := der;
    a[medio] := x;
  end; { fin de dividir }
```

Este último proceso requiere n-1 comparaciones y a lo sumo n+1 intercambios para dividir el vector de n elementos en dos partes. En el caso promedio, el vector queda subdividido en dos de igual tamaño, de donde el `QuickSort` requiere en promedio, un número de comparaciones e intercambios proporcional a $n\log_2(n)$, al igual que el *sort by merge*. Dado que el algoritmo `Dividir` realiza menos intercambios que el *merge*, el `QuickSort` resulta ser más eficiente que el *sort by merge*.

Desafortunadamente, en el peor caso, el `QuickSort` resulta ser muy ineficiente. Este peor caso ocurre cuando la lista a ser ordenada ya se encuentra ordenada en forma previa, por lo tanto, el ítem que se encuentra a la derecha ya está en su lugar y la recursión, en lugar de ordenar subvectores cada vez más chicos trabaja sobre un subvector de un único elemento y otro muy grande. Se puede pensar en mejorar esta situación, seleccionando el elemento inicial del proceso Dividir de manera adecuada, pero de todas formas, siempre existirán casos donde el `QuickSort` emplee una cantidad de comparaciones proporcional a n^2.

A continuación se presenta el código del método de ordenación `QuickSort` para ordenar un vector de longitud n.

Código

```
procedure  Sort( var  a: tvector;      {vector a ordenar}
                      n: indice);      {cantidad de elementos}
  { aquí iría el procedure Dividir }
  procedure QuickSort( var a: tvector; ini, ult: indice );
    { ordena a[ini], ..., a[fin] de manera recursiva }
    var medio: indice;
    begin
      if ini < ult then
      begin
        Dividir(a, ini, ult, medio);
        QuickSort(a, ini, medio-1);
        QuickSort(a, medio, ult);
      end;
    end;  {QuickSort}
  begin
    QuickSort( a, 1, n);
  end;
```


8.11 Recursividad y eficiencia

La **recursividad** es una herramienta muy poderosa que puede dar soluciones muy claras a problemas complejos. Pero tiene algunas desventajas. Desde el punto de vista del tiempo insumido, una solución no recursiva puede ser más eficiente que otra recursiva y esto se debe a:

- El *overhead* asociado con la llamada a subprogramas.
- La ineficiencia inherente de algunos algoritmos recursivos.

El primer factor no es privativo de los algoritmos recursivos, pero dado que su funcionamiento es mediante llamadas a sí mismo, este problema se ve amplificado. Por ejemplo, el cálculo de Factorial(n) produce n llamadas recursivas.

Como contrapartida, la recursividad permite la obtención de soluciones más claras.

El verdadero uso de la recursión es el de herramienta para resolver problemas para los cuales no existe una solución iterativa sencilla.

El segundo punto a remarcar es la ineficiencia inherente de los algoritmos recursivos. Aquí la ineficiencia no tiene que ver con el *overhead* natural de los algoritmos recursivos, sino con la forma en que el algoritmo resuelve el problema.

Analizando nuevamente el problema de los conejos:

Ejemplo 8.28

```
Function Conejos(n: word): word;
  { n es un entero positivo }
  begin
    if n <= 2
      then Conejos := 1
      else Conejos := Conejos(n-1) + Conejos(n-2);
  end;
```

Es importante ver el funcionamiento del algoritmo cuando se busca calcular Conejos(5) (Figura 8.1)

Como se puede ver, el problema fundamental de esta función es que los mismos valores son calculados una y otra vez. Por ejemplo, Conejos(3) es calculado dos veces y Conejos(2), tres veces.

Cuando n es grande, la cantidad de veces que se calcula lo mismo crece considerablemente.

Queda como ejercicio para el lector la implementación del algoritmo iterativo para el mismo problema (use la misma idea del recursivo).

Conclusiones

El término eficiencia no debería relacionarse únicamente con el tiempo de ejecución de un proceso, sino con la buena administración de **todos** los recursos disponibles, entre los que se halla el tiempo de procesamiento. Hay que tener en cuenta que según del tipo de aplicación de que se trate, puede ser importante que el algoritmo haga una buena administración de otros recursos de la máquina como lo es la memoria.

La recursividad es una técnica de programación muy útil para resolver problemas en base a su definición. La eficiencia de los algoritmos recursivos se nota en aplicaciones complejas. Se discutirá nuevamente este tema cuando se presente o estudien las estructuras no lineales como los grafos y los árboles.

Se han mostrado algunas técnicas para poder evaluar el tiempo de ejecución de un algoritmo, sea o no recursivo.

Ejercicios

En todos los casos, escribir el algoritmo y analizar su eficiencia (tiempo y memoria requerido).

1. Convertir una matriz de números de dimensión N*M en un vector ordenado de menor a mayor.
2. Dada una matriz A de números, con dimensión N*M, obtener otra matriz A´ con las filas ordenadas de menor a mayor según la suma de sus valores.
3. Dada una matriz de N*M con datos de clientes (nombre y apellido, dirección, edad y saldo) discuta el modo de poder acceder a todos los datos, ordenados por saldo o por nombre y apellido.
4. Dadas las coordenadas de dos puntos (x1,y1) (x2,y2) desarrollar un algoritmo recursivo que trace el segmento que los una, utilizando solo coordenadas enteras (Sugerencia: dibuje puntos, comenzando por el punto medio). Analice el tiempo y la memoria dibujando el segmento con 10, 100 y 1.000 puntos.

9

Capítulo 9

Datos compuestos enlazados: listas, árboles y grafos

9

Datos compuestos enlazados:
listas, árboles y grafos

Objetivo

Se presentan las estructuras de datos dinámicos de mayor interés: listas, árboles y grafos. Interesa mostrar la vinculación de los datos mediante punteros y el manejo dinámico de la memoria en el caso de la estructura lineal lista.

Árboles y grafos son presentados en un modo introductorio, para que el lector se familiarice con estructuras de datos no lineales.

Se discuten ejemplos de algoritmos clásicos de recorrido por las distintas estructuras, analizando alternativas de representación de los datos.

Es importante que el lector asocie determinadas estructuras de datos con clases de problemas, de modo de incorporar el concepto de "tipo de dato orientado a la aplicación".

9.1 Listas como estructura de datos

Se sabe intuitivamente lo que es una lista, en la vida diaria se utilizan en forma constante como, por ejemplo, lista de compras, lista de libros, lista de alumnos, etc. En programación las listas son un tipo de dato muy útil y de uso frecuente. Desde el punto de vista de la programación una lista es una colección de elementos homogéneos, con una relación lineal que los vincula. Esto significa que cada elemento menos el primero tiene un único predecesor y cada elemento menos el último tiene un sucesor en la lista. El orden de vinculación de los elementos de la lista afecta a su función de acceso.

Las estructuras de datos lineales que se han visto se pueden clasificar en estáticas y dinámicas.

En capítulos anteriores se han tratado estructuras estáticas tales como arreglos y registros, para los cuales, una vez dada la declaración de su estructura queda definida su forma y ubicación en memoria. Dicha memoria permanecerá ocupada mientras el procedimiento que incluye la declaración esté activo.

Entre las estructuras dinámicas que varían en tamaño y en forma, se analizaron las pilas y colas, en las cuales la ocupación de memoria se modifica continuamente según las operaciones de *push* y *pop* realizadas.

Ahora, se presenta el problema de estructuras de datos con acceso secuencial tales como arreglos, pilas y colas (Figura 9.1).

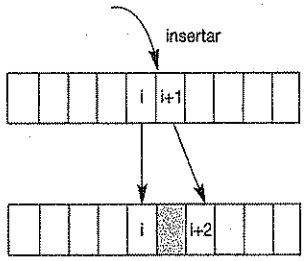


Figura 9.1

Si se analizan brevemente las operaciones que se pueden realizar con ellas, se encuentra la de intercalar un elemento en medio de una estructura de cualquiera de los tipos mencionados es relativamente complejo. Esto se debe principalmente a la restricción de que los elementos sucesivos de la estructura ocupan posiciones consecutivas de la memoria; entonces, intercalar significa "correr" un conjunto de datos para permitir "abrir" el lugar para el nuevo elemento y recién luego es posible incorporarlo.

A continuación se analizará el tipo lista que es una estructura lineal dinámica con un manejo muy flexible de la operación de intercalado.

BIBLIOTECA
FAC. DE INFORMATICA
U.N.L.P.

Definición

Una lista dinámica es un conjunto de elementos de tipo homogéneo, donde los mismos no ocupan posiciones secuenciales o contiguas de memoria, es decir, los elementos o componentes de una lista pueden aparecer físicamente dispersos en la memoria, si bien mantienen un orden lógico interno.

9.1.1 Punteros

Como los elementos que forman una lista, no están en posiciones consecutivas de memoria, para poder trabajar con ellos se necesita direccionarlos. Por esto, cada elemento de la lista contiene un indicador, que apunta al próximo elemento de la misma. Es decir, se requiere un nuevo tipo: el tipo puntero.

Se debe recordar que las diversas posiciones de almacenamiento en la memoria de una máquina se identifican mediante direcciones numéricas; si se conoce la dirección de un dato se lo puede ubicar directamente. Así después de almacenar un dato en una celda de la memoria, se puede almacenar la dirección de ese dato en otra; luego si se quiere recuperar el dato y se tiene acceso a la celda que contiene la dirección, se puede consultar el dato de dicha dirección.

Definición

Un puntero es un tipo de variable, en el cual se almacena la dirección de un dato y permite manejar direcciones "apuntando" a un elemento determinado.

Con la utilización adecuada de punteros, se puede considerar una lista como constituida por elementos lógicamente adyacentes.

Cada elemento de la lista se puede representar como se muestra en la Figura 9.2.

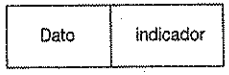


Figura 9.2

El campo dato es la información que se maneja, el que puede estar constituido por un conjunto de elementos de algún tipo previamente definido.

El campo indicador es una variable de tipo puntero, cuyo contenido es la dirección del próximo elemento de la estructura.

Para crear una lista utilizando punteros, se debe crear cada uno de sus nodos para luego enlazarlos con algún criterio.

Para ello, se solicita espacio en la memoria donde se almacenará la información. Ese pedido de lugar en la memoria se hace por medio de la instrucción:

```
new ( Var_de_tipo_puntero )
```

y la dirección de la memoria asignada queda cargada en `Var_de_tipo_puntero`.

Por lo tanto, cada elemento de la lista será una variable tipo *record* pues, además de definir todos los campos que pertenecen al dato, se requiere una variable de tipo puntero que permitirá el "enganche".

Ejemplo 9.1

En Pascal, la definición del elemento de la lista será de la forma:

```
type ppun = ^reg;
     reg  =  record
             nom  : string[50];
             edad : integer;
             sig  : ppun ;
     end;
```

ppun, así definido, dice que toda variable de este tipo es un puntero el cual apunta a alguna variable de tipo `reg`. Si se declara:

```
var  p1,p2: ppun;
     reg1,reg2 : reg;
```

Tanto p1 como p2 podrán, por ejemplo, contener direcciones de las estructuras `reg1` y `reg2`. Además `reg1.sig` y `reg2.sig` son también punteros a variables de tipo `reg`.

Por ejemplo, si se arma una lista con datos de tipo `reg`, se la puede encadenar utilizando sucesivamente `reg1.sig`.

Para realizar operaciones con el campo nom del registro apuntado por `p1`, debe escribirse: `p1^.nom` donde `p1` es una variable tipo `ppun`.

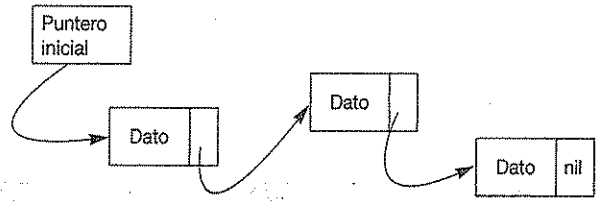


Figura 9.3

Para saber dónde está la primer entrada de una lista, se guarda en una celda de la memoria la dirección de la primera entrada. Esta celda apunta al principio de la lista y suele llamarse puntero inicial. Si se quiere leer la lista, se comienza en la posición indicada por este punteo y se halla el primer nombre junto con un puntero a la siguiente entrada. Si se sigue este puntero, se halla la segunda entrada, y así sucesivamente hasta el final de la lista. Figura 9.3

Para detectar el final de la lista se utiliza un puntero nulo (NIL) que es un patrón especial que aparece en la celda de puntero en la última entrada y que indica que no hay más entradas en la lista.

Ejemplo 9.2

Realizar un algoritmo que lea datos personales y genere con ellos una lista.

Precondición
- Lista vacía
- Puntero inicial en NIL

Postcondición
- Datos cargados en la lista
- El puntero inicial apunta al primer dato
- El puntero final esta en NIL

```
Program Crea_Lista;
   type ppun  = ^reg;
        reg = record
              nom: string[50];
              edad: integer ;
              sig: ppun ;
        end;
   var pri, nue, ult: ppun;
       nom1: string[50] ;
   begin
     pri := NIL;
     write ( 'Nombre : ' );
     readln ( nom1) ;
     while nom1<>'FIN' do
       begin
         new(nue);
         nue^.nom  := nom1;
         write( ' tiene ' );
         readln(nue^.edad) ;
         writeln;
         if pri = NIL
           then  pri := nue
           else  ult^.sig := nue;
         ult := nue ;
         ult^.sig := NIL ;
           write ( 'Nombre : ' );
           readln (nom1) ;
       end;
   end;
```

Se observa que hay un puntero más: pri. Dicho puntero se utiliza para guardar la dirección inicial de la lista. En otras palabras, el contenido de la variable `pri` es la dirección del primer elemento de la estructura de datos.

El siguiente segmento de algoritmo presenta las instrucciones necesarias para recorrer la lista creada en el programa anterior.

```
nue := pri ;
while (nue <> NIL) do
   with nue^ do
     begin
       writeln ( nom , 'tiene' , edad );
         nue := sig ;
     end;
```

9.1.2 Operaciones con listas

Se presentan a continuación algunas operaciones que se pueden realizar con listas:

- Recorrer una lista.
- Acceder al k-ésimo elemento de la lista.
- Intercalar un nuevo elemento en la lista.
- Agregar un elemento al final de la lista.
- Combinar dos listas, para formar una sola.
- Localizar un elemento de la lista para conocer su posición dentro de ella.
- Borrar un elemento de la lista.

9.1.2.1 Borrar un elemento de la lista

La Figura 9.4 presenta gráficamente la operación consistente en borrar un elemento de una lista, eliminando el `dato3`. Esto significa que se altera un apuntador. El puntero anterior, que apunta al elemento a borrar debe apuntar al que será el elemento siguiente.

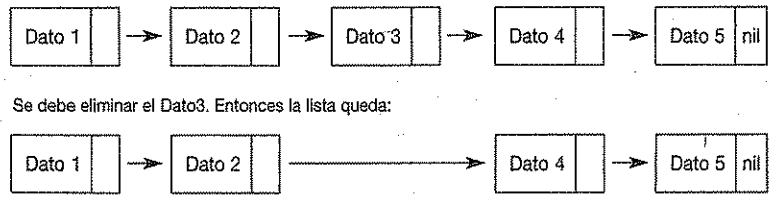


Figura 9.4

Ejemplo 9.3

Utilizando la lista generada anteriormente, el algoritmo Pascal para borrar un elemento queda como sigue:

```
{ Nom1 contiene el nombre a borrar }
read (nom1) ;

{Inicializo las variables tipo puntero nue y ult }
nue := pri;
ult := pri;
while  (nue <> NIL) do
  begin
    if (nue^.nom = nom1) then
      begin
        if (nue = pri) then
          begin
            pri := nue^.sig;
            ult := pri;
          end
          else ult^.sig:=nue^.sig;
      end
      else ult := nue;
    nue := nue^.sig;
  end;
```

Se dejan como ejercicio para el lector los siguientes casos:

- El nombre a borrar aparece más de una vez
- Se desea borrar todos los empleados cuya edad sea mayor que 49 años.

9.1.2.2 Agregar un elemento al final de la lista

Para lograrlo, se ubica el puntero al final de la lista, se lee el dato nuevo y se lo agrega haciendo los enganches correspondientes.

Código

```
{ ubicarse al comienzo de la lista }
nue := pri ;
{ se recorre la lista hasta el final }
while  ( nue <> NIL ) do
   begin
      ult := nue ;
      nue := nue^.sig ;
   end;

{ instancia el puntero, se lee el dato y luego se agrega }
```

```
new(nue);
read( nue^.nom);
write(' tiene ');
readln(nue^.edad) ;
ult^.sig := nue ;
nue^.sig := NIL;
```

9.1.2.3 Intercalar un elemento en la lista

Se tiene el caso una lista ordenada de acuerdo a algún criterio y se debe agregar un nuevo elemento de forma tal que la lista continúe ordenada, realizando los enganches lógicos correspondientes. Tal como se presenta en la Figura 9.5.

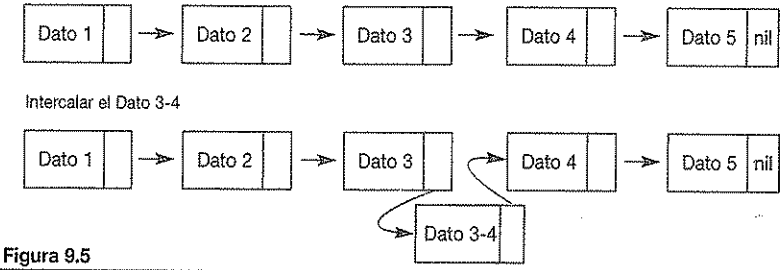


Figura 9.5

Esto significa cambiar el enganche del elemento anterior a la posición a intercalar y enganchar dicha posición con el elemento siguiente.

Código

```
{ se pide espacio y se lee el nuevo dato }
new(nue1);
read( nue1^.nom);
write(' tiene ');
readln( nue1^.edad );
{ se inicializa los punteros }
nue := pri;
ult := pri;
{Se recorre mientras no se termine y no se encuentre la posición correcta}
while (nue<>NIL) and (nue^.nom < nue1^.nom1) do
   begin
      ult := nue;
      nue := nue^.sig ;
   end;
   if (nue = ult)
      then pri := nue1
      else ult^.sig  := nue1;
   nue1^.sig := nue ;
```


9.1.2.4 Acceder al k-ésimo elemento de la lista

Para ello se necesita como parámetros a: k que indicará la posición del elemento de la lista, pri que tiene el comienzo de la lista y la variable nom2 que contendrá el nombre del elemento que está en la posición k. El conjunto de instrucciones es el siguiente:

Código

```
if (pri = NIL)
   then write ('ERROR - La lista no contiene elem.')
   else begin
    {se inicializa un contador para las posiciones, se ubica al
     principio de la lista}
      pos := 1;
      nue := pri;
      { se recorre la lista  }
      while ( pos<k ) and ( nue<>NIL ) do
         begin
            pos := pos + 1;
            nue := nue^.sig;
         end;
      if (pos=k) then
         begin
            nom2 := nue^.nom;
            write ('el nombre es : ',nom2);
         end
         else write(k,' excede la longitud.');
   end;
```

9.1.2.5 Localizar la dirección de un elemento dado.

Para ello se tiene: la variable pri que indica el comienzo de la lista, nom2 variable que contiene el nombre cuya dirección de memoria se desea ubicar y pos que es la variable que contendrá la dirección de nom2. El segmento de código quedaría como sigue:

Código

```
if (pri=NIL)
   then write ('ERROR')
   else
   begin
      { ubicarse al principio de la lista }
      nue := pri;
      while (nue<>NIL)  and  (nom2<>nue^.nom ) do
         { Avanzo en la lista }
         nue := nue^.sig;
         if (nue<>nil nom=nom2)
            then pos:= nue            {Guardo la dirección}
            else {Se acabó la lista y no encontré el nombre}
      write('NO HALLADO')
   end;
```

Listas Circulares

Las listas enlazadas no permiten a partir de un elemento acceder directamente a cualquiera de los elementos que la preceden. En lugar de almacenar un puntero **nulo** se hace que el último elemento apunte al primero o principio de la lista, este tipo de estructura se llama lista circular.

Figura 9.15

Las listas circulares presentan las siguientes ventajas respecto de las listas simples.

- Cada nodo de la lista circular es accesible desde cualquier otro nodo de ella.
- Las operaciones de concatenación y división de las listas son más eficaces con listas circulares.

El inconveniente es:

- Se pueden producir bucles o lazos infinitos. Una forma de evitar estos bucles es disponer de un nodo especial denominado cabecera que se puede diferenciar de los otros nodos por tener un valor especial.

Listas doblemente enlazadas

En las listas anteriormente mencionadas solo de podía realizar el recorrido en un único sentido (del principio hasta el final). En numerosas ocasiones se necesita recorrer la lista en ambos sentidos. Las listas que se pueden recorrer en ambas direcciones se llaman listas doblemente enganchadas o enlazadas. En este tipo de listas, además del campo información se cuenta con dos variables del tipo puntero que apuntan al nodo anterior y siguiente. Obviamente, una lista doblemente enlazada ocupa más espacio de memoria que una lista simple para la misma cantidad de información.

9.1.3 Conclusiones sobre listas

Se pueden efectuar las siguientes conclusiones:

- Una lista es una estructura lineal dinámica.
- Los datos **no** necesariamente residen en posiciones consecutivas de la memoria.
- La mayor facilidad que brindan las listas es el "enganche" dinámico de datos a través de variables auxiliares de tipo puntero, que permiten un fácil intercalado o borrado de información, sin un movimiento masivo de datos en la memoria.

- En muchas ocasiones, dada una estructura de datos con gran cantidad de información, organizada secuencialmente (por ejemplo, un archivo), se puede asociar una o más "listas" que indican diferentes secuencias de acceso a la estructura por diferentes campos. De esta manera, sin reordenar la información, se la puede manejar como estructurada y organizada "por claves" o por diferentes criterios, según la lista que se utilice como "guía", como se puede ver en la Figura 9.6.

```
type ppunt = ^reg
     reg=record
         nombre: string[50];
         edad: integer;
         sig: ppunt;
     end;
```

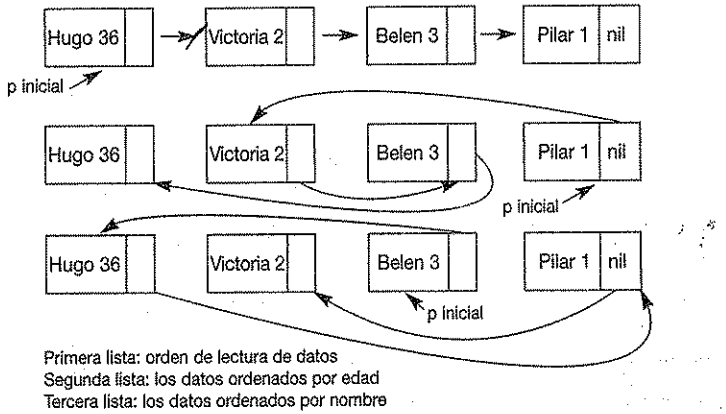


Figura 9.6

9.2 Árboles

Una lista lineal esencialmente tiene un sucesor para cada elemento. Hasta ahora, no se cuenta con una estructura de datos que permita representar jerarquías como, por ejemplo, las relaciones que existen en una empresa o la representación de una red ferroviaria (más de un sucesor por elemento).

En esta sección se presenta una estructura, conocida como árbol, que es apropiada para la representación de relaciones jerárquicas como las que existen entre los miembros de una familia, entre los que trabajan en una misma empresa o entre los módulos de un programa.

Los árboles permiten realizar aplicaciones muy variadas en el área informática:

- Es posible desarrollar algoritmos de búsqueda eficientes representando la secuencia de pruebas de una búsqueda binaria como un árbol.
- Pueden organizarse archivos en una computadora utilizando un árbol de directorios: un directorio principal contiene información de varios subdirectorios (por ejemplo, un directorio para la materias del área de Informática y otro para las materias de Matemáticas) y cada uno de estos puede contener tanto archivos como otros subdirectorios (por ejemplo, el programa de cada asignatura).
- Es posible evaluar expresiones aritméticas y ejecutar programas en Pascal representando la sintaxis de estas expresiones y programas como un árbol.
- Es posible determinar quién es el ganador de un juego de estrategia representando las posiciones permitidas en el juego mediante un árbol.

Definición

Un árbol es una estructura de datos que satisface tres propiedades: (1) cada elemento del árbol (nodo) se puede relacionar con cero o más elementos, los cuales llama "hijos"; (2) si el árbol no está vacío, hay un único elemento al cual se llama raíz y que no tiene padre (predecesor), es decir, no es hijo de ningún otro; (3) todo otro elemento del árbol posee un único padre y es un descendiente (hijo del hijo del hijo, etc.) de la raíz.

Los árboles se representan mediante esquemas como el que muestra la Figura 9.7, donde pueden verse líneas que conectan a cada elemento con sus hijos. La raíz aparece en el tope del árbol y, a diferencia de los árboles que se conocen habitualmente, estos "crecen" hacia abajo. Cada elemento del árbol se denomina nodo y los nodos que no tienen hijos (la mayoría se encuentran en la parte más baja del árbol) son las hojas del árbol. Un camino en un árbol es una secuencia lineal de elementos, donde cada uno es el padre del próximo ítem de la secuencia. Una rama en un árbol es un camino que se extiende desde la raíz hasta una hoja. Un subárbol de un árbol es un nodo con todos sus descendientes. El nivel de un nodo en el árbol es el número de ítems que hay sobre el único camino que lo une con la raíz. La altura de un árbol es el máximo entre los niveles de los nodos del árbol.

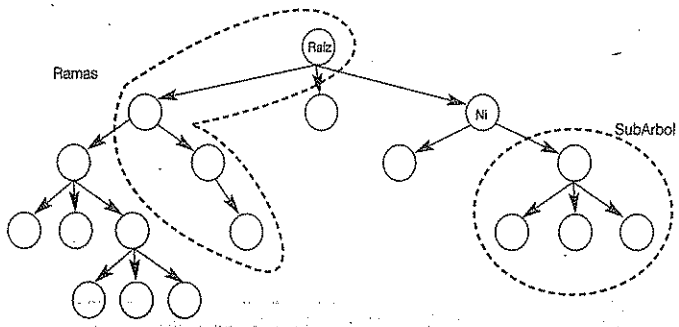


Figura 9.7

Definición

Un árbol donde cada nodo no tiene más de dos hijos se denomina árbol binario.

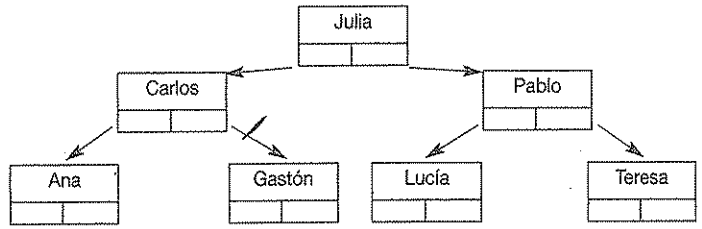


Figura 9.8

Un árbol binario se dice que está ordenado cuando cada nodo del árbol es mayor (según algún criterio preestablecido) que los elementos de su subárbol izquierdo (el que tiene como raíz a su hijo izquierdo) y menor o igual que los de su subarbol derecho (el que tiene como raíz a su hijo derecho).

Como puede verse, una pequeña variante del método de búsqueda binaria o dicotómica permite hallar un elemento en un árbol binario ordenado.

El elemento buscado se compara con la raíz y si es menor se repite el proceso comparando contra el hijo izquierdo en caso contrario continuamos por la derecha. El proceso termina cuando se encuentra el elemento o cuando se termina la rama.

9.2.1 Representación de árbol binario

Para el manejo de árboles se utiliza una estructura basada en punteros como se vio en las listas. Se puede utilizar la siguiente representación:

Código

```
type tipoElem : ... ;
        arbolBinario = ^nodo;
        nodo = record
            elem: tipoElem;
            izq, der: arbolBinario;
        end;
var arbol: arbolBinario;
```

En este caso se utilizan dos punteros por cada nodo y, sin embargo, al igual que ocurría con las listas, los árboles son (en base a esta representación) una estructura de datos dinámica. Esto permite insertar y borrar elementos dentro del árbol sin necesidad de mover otros nodos.

9.2.2 Conclusiones

Las estructuras de datos árbol y árbol binario son los primeros ejemplos de estructuras no lineales.

Desde el punto de vista algorítmico se verá que las operaciones sobre árboles requerirán algoritmos recursivos, que han de resultar más claros y eficientes.

9.3 Grafos

Las estructuras de datos que se han utilizado hasta ahora, pueden re-clasificarse según el tipo de proceso que se realice con ellas como:

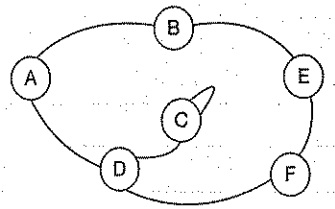
- Estructuras lineales: aquellas donde los elementos están organizados en forma secuencial con un sucesor y un predecesor. A esta clase pertenecen: pilas, colas, arreglos y matrices y listas, todas estructuras cuya recorrida es esencialmente secuencial.
- Por otro lado, se encuentran las estructuras no lineales en las que un elemento puede tener más de un sucesor o predecesor: en general acceden a sus elementos según un criterio determinado, es decir, no secuencial. A esta clase pertenecen los árboles n-arios y en particular los árboles binarios.

Existen algunas aplicaciones en las cuales es necesario representar una relación de muchos a muchos, como ocurre con las ciudades que se comunican a través de distintos caminos o un sistema distribuido donde pueden existir distintas alternativas para conectar dos computadoras.

Para este tipo de aplicaciones, es necesario utilizar una estructura de datos no lineal, que llamaremos grafo.

Definición

Un grafo G = (V, A) está formado por un conjunto de vértices V, y un conjunto de arcos, A; cada arco se presenta mediante un par (v, w), donde v y w ∈ V. Ver Figura 9.9.

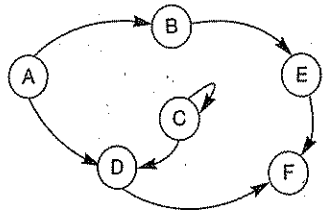


G= (vértices, Arcos)
Arcos= {(A,B), (A,D), (B,E), (C,C), (C,D), (D,F), (E,F)
Vertices= {A, B, C, D, E, F}

Figura 9.9

Definición

Si el par está ordenado se dice que el grafo es dirigido. Los grafos dirigidos generalmente se llaman digrafos. Figura 9.10.



G= (vértices, Arcos)
Arcos= {(A,B), (A,D), (B,E), (C,C), (C,D), (D,F), (E,F)}
Vertices= {A, B, C, D, E, F}

Figura 9.10

Definición

Se dice que el vértice w es adyacente a v si y solo si (v,w) ∈ A. En un grafo no dirigido con arco (v,w) y, por consiguiente, (w,v), w es adyacente a v y viceversa.

Definición

Un camino en un grafo es una secuencia de vértices, $w_1, w_2, ..., w_n$, tal que $(w_j, w_j+1) \in A$, para $1 \leq j < n$. La longitud de un camino es la cantidad de arcos que contiene. Es posible obtener un camino que comience y termine en el mismo vértice. Si este camino tiene longitud 1, es decir, el grafo tiene arcos del tipo (v,v), se dice que el grafo tiene un *loop*. En general se trabajará con grafos donde esta última situación no ocurre.

Definición

Un camino simple es aquel donde todos sus vértices son distintos. Solo el primero y el último pueden coincidir.

Definición

Un ciclo, en un grafo dirigido, es un camino de longitud mayor o igual que 1 donde el primer y el último vértice es el mismo; $w_1 = w_n$. Este ciclo se llamará ciclo simple si el camino es simple. En caso de grafos no dirigidos, es necesario que los arcos sean distintos. La idea es que el camino u, v, u en un grafo no dirigido no puede ser considerado un ciclo ya que se trata de un solo arco, (u,v) es el mismo que (v,u). En un grafo dirigido se trata de distintos arcos.
Figura 9.11.

Definición

Un grafo es acíclico si no tiene ciclos.

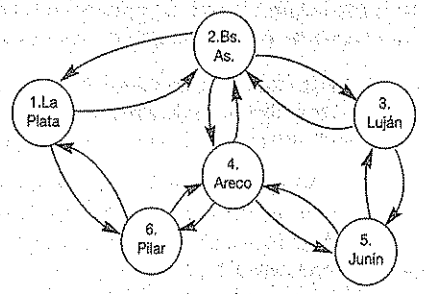
En algunas aplicaciones, los arcos tienen una tercera componente que es su peso o su costo.

Definición

Un grafo no dirigido está conectado si para cualquier par de vértices existe un camino que los une. Un grafo dirigido con esta propiedad se dice que está fuertemente conectado. Si un grafo dirigido no está fuertemente conectado, pero el grafo subyacente (sin dirección en los arcos) está conectado, se dice que es débilmente conectado.

Ejemplo 9.4

Un ejemplo del mundo real podría ser la modelización de un sistema de aeropuertos mediante un grafo. Cada aeropuerto sería un vértice del grafo y dos vértices estarían conectados por un arco si existe un vuelo sin escala de uno a otro. Dicho arco podría tener un peso (representando, por ejemplo, costo, distancia o tiempo del vuelo).



Arcos= {(1,2), (2,1), (2,3), (3,2), (3,5), (5,3), (1,6), (6,1), (4,6), (6,4), (2,4), (4,2), (4,5), (5,4)}

Vertices= {1, 2, 3, 4, 5, 6}

Camino= {1 2 3 5}

Longitud= 4

Ciclo= {1 2 3 5 4 1}

Loops= no existe

Adyacentes= 1 es adyacente de 2 y 6
2 es adyacente de 1,4 y 3
3 es adyacente de 2 y 5
4 es adyacente de 2, 5 y 6
5 es adyacente de 3 y 4
6 es adyacente de 1 y 4

Figura 9.11

Sería razonable que el grafo fuese dirigido permitiendo, de esta manera, que el costo del vuelo varíe dependiendo de las tarifas locales.
También se podría estar interesado en que el aeropuerto estuviera fuertemente conectado permitiendo de esta forma volar directamente de un lugar a otro.
En estos casos suele ser interesante conocer, por ejemplo, cuál es la ruta más rápida para llegar a destino o cuál es la más económica, con lo que se ve la necesidad de disponer de algún criterio para obtener el "mejor" camino.

9.3.1 Representación de Grafos

Se consideraran solo grafos dirigidos (los no dirigidos son similares).

Una manera simple de representar un grafo es utilizar una matriz. Esta estructura es conocida como matriz de adyacencia.

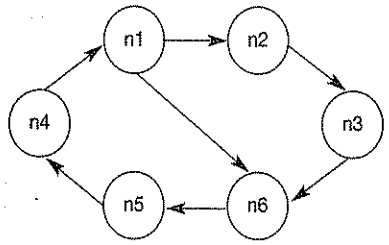
Para cada arco (u, v), se asigna M[u, v] := 1 (como forma de indicar que el arco existe) y en caso contrario, el valor de la matriz es cero. Si el arco tiene un peso asociado, puede asignarse `M[u,v]:=` peso y utilizar un valor centinela, muy grande o muy chico para indicar que no existe arco.

Por ejemplo, si se busca el camino de menor costo en el modelo del sistema de aeropuertos, se puede representar la no existencia de vuelo mediante un costo ∞. Por el contrario, si se busca el camino más caro se puede utilizar -∞ o 0 como forma de indicar que no existe vuelo.

Esta representación tiene el mérito de ser extremadamente simple, sin embargo, el espacio necesario para guardarla en memoria, es proporcional a N^2, siendo N la cantidad de vértices. Nótese que el tamaño de memoria es independiente de la cantidad de arcos. Esto puede redundar en un tamaño totalmente prohibitivo.

Supongamos 1.000 destinos N x N = $N^2 = 10^6$ posiciones en memoria.

Una matriz de adyacencia es la representación adecuada en caso de tener un grafo denso, es decir, cuando la cantidad de arcos se acerca a N2. Pero, en general, esto no se cumple. La Figura 9.12 muestra un grafo y su matriz de adyacencias correspondientes.



Matriz de adyacencia

0	1	0	0	0	1
0	0	0	1	0	0
1	0	0	0	0	0
0	0	0	0	0	1
0	0	0	1	0	0
0	0	0	0	1	0

M[ij]=1 si existe un arco que sale de ni y llega a nj, sino vale 0

Figura 9.12

Cuando el grafo es esparcido, es decir, que tiene pocos arcos, la mejor solución es usar una lista de adyacencias. En esta representación, para cada vértice se mantiene una lista con todos sus adyacentes. Por lo tanto, el espacio requerido será la suma del total de arcos y el total de vértices. Ver Figura 9.13.

Como se puede ver, la estructura que se encuentra más a la izquierda no es más que un arreglo de punteros a listas.

La lista de adyacencia es la manera normalizada de representar un grafo. Los grafos no dirigidos pueden representarse de la misma forma, solo que cada arco (u,v) aparecerá en dos listas, con lo cual, el espacio utilizado será el doble.

Las operaciones más comunes sobre grafos son:

- Saber si existe un camino que una dos vértices dados. En nuestro ejemplo, es saber si existe un vuelo de 'Buenos Aires' a 'México'.
- Saber cuál es el mejor camino, en general. utilizando arcos con peso, donde el término "mejor" puede ser: más corto, más barato, más rápido, etc.

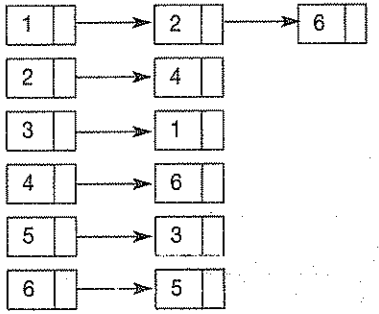


Figura 9.13

Ejemplo 9.5

Se intentará abordar el primer problema y para ello se verá el siguiente ejemplo. La Figura 9.14 muestra la distribución de nueve aeropuertos con sus vuelos directos correspondientes.

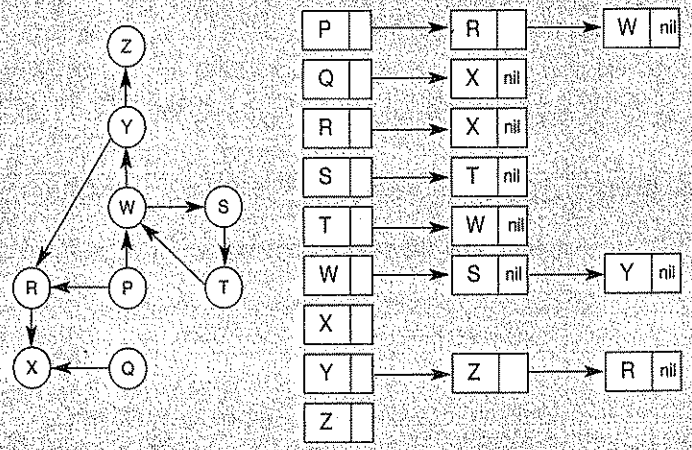


Figura 9.14

9.3.2 Buscando un camino entre dos vértices del grafo

Para saber si existe un camino que una dos vértices dados se realiza una búsqueda exhaustiva. Esto es, comenzando por la ciudad de origen, se seguirán todos los caminos posibles hasta que se llegue a destino o hasta que se los haya recorrido todos.

Lo que se hace es tomar un adyacente al vértice de partida, lo que conducirá a la ciudad C1 y si es la que se busca, termina el proceso, sino se reintenta con un adyacente de C1. Así se llega a C2. Si C2 era el punto de llegada, se finaliza, sino se sigue repitiendo el proceso.

Se consideran las posibles salidas de la estrategia presentada:

1. Se logra hallar el camino que una los dos vértices indicados.
2. Se llega a una ciudad que no tiene adyacentes.
3. Se quedan dando vueltas en círculos.

Se ve que no necesariamente solo ocurre la opción 1), que es la que realmente interesa. Puede ocurrir que no exista un camino entre los dos vértices como ocurre con P y Q. Es más, puede existir el camino y aun así ocurrir las opciones 2) o 3). Se verá qué pasa si se quiere obtener un camino entre P y Z. Con este algoritmo, se puede quedar en el ciclo W,S y T eternamente. Nótese que llegar a 2) no significa que no exista el camino.

Se supone una vez más que se quiere hallar un camino entre P y Z. El algoritmo presentado seleccionará como primer opción a R y luego a X. Cuando no haya más adyacentes para continuar, se debe poder retroceder y retornar a la ciudad anterior a X. Al volver a R y ver que no tiene más adyacentes que X, que ya se visitó, se debe poder seguir retrocediendo para poder seleccionar otro camino. Esto implica que se debe guardar información respecto al orden en que las ciudades fueron visitadas para poder hacer el camino inverso.

9.3.2.1 Solución utilizando una pila

Una opción podría ser guardar la secuencia de ciudades en un pila. Cada vez que se va a visitar una ciudad, se la apila y cuando se quiere volver a la ciudad anterior, se desapila. Se puede escribir el algoritmo siguiente:

Código

```
Push( Pila, origen)
while (no haya determinado si hay una secuencia de vuelos del
   origen al destino) do
    begin
        if es necesario volver a la ciudad anterior
            then Pop( Pila, ciudad)
            else begin
                seleccione un ciudad C adyacente a la ciudad que se
                    encuentra en el tope de la pila
                push( Pila, C )
            end
    end
```


Como se puede ver, en el tope de la pila se halla la ciudad que actualmente se está visitando. Se puede ver que la pila siempre conserva el camino de origen a la ciudad actual. Si se logra llegar a destino, indicará el camino que se siguió.

Falta determinar, de una manera precisa, en qué casos es necesario volver a la ciudad anterior.

Las posibilidades son: que la ciudad que se está visitando no tenga más adyacentes o que ya se haya pasado por esta ciudad. La condición de no visitar más de una vez a una ciudad lo libera de los ciclos. Como conclusión, es necesario volver atrás (desapilar) cuando ya no queden ciudades sin visitar adyacentes a la que se encuentra en el tope de la pila. Para reconocer si una ciudad ha sido visitada o no, se la debe marcar.

Queda por analizar la condición de fin del algoritmo. Ya se ha visto que la pila contiene el camino desde el origen hasta la ciudad actualmente visitada, por lo tanto, si la pila logra en algún momento tener en el tope la ciudad destino, se ha hallado el camino buscado. En caso de que no exista tal camino, las ciudades se irán desapilando hasta que todas sean visitadas. Cuando no queden más adyacentes de la ciudad origen sin visitar, la pila quedará vacía.

Se puede reescribir entonces el algoritmo de la siguiente forma:

Código

```
BuscarCamino( Origen, Destino )
        {ponemos la ciudad origen en la pila}
        push( pila, origen)
        marcar el origen como visitado
 while not (st_Empty( pila )) and  (st_Top( pila ) <> destino) do
   begin
      if no quedan adyacentes de la ciudad que se encuentre en el
      tope de la pila sin visitar
        then st_Pop( pila, ciudad)
        else begin
           Seleccione un ciudad C adyacente a la ciudad que se
            encuentra en el tope de la pila
              st_Push( pila, C )
           marcar C como visitada
        end
   end
```

Nótese que el orden de selección de las ciudades a visitar no está determinado por el algoritmo sino por la manera en que se construyó el grafo. De todas formas, esto no afecta el resultado final.

Se probará el algoritmo con un ejemplo para comprender definitivamente como funciona: se necesita hallar un camino entre P y Z. (Tabla 9.1)

9.3.2.2 Solución utilizando un algoritmo recursivo

A partir del vértice de origen, se toma el primer adyacente que no se hubiese visitado, si era el vértice de destino, el proceso termina, si no se sigue buscando a partir de él. Este proceso se repite hasta que se encuentre el destino o hasta que se recorre todo el grafo. El algoritmo se puede escribir de la siguiente forma:

Código

```
BuscarCaminoR( Origen, Destino )
    { Busca un camino entre Origen y Destino}
    marcar Origen como visitado
    if Origen = destino
       then Se termina el proceso, se llego a destino
       else
            Para cada vértice C adyacente a Origen, no visitado
            BuscarCamino( C, Destino)
```

Nótese que el uso de un algoritmo recursivo simplifica la implementación ya que no es necesario utilizar una pila para recordar el camino andado. El proceso se va "apilando" en forma automática, de manera que al terminar, ya sea porque encontró el vértice destino o porque no hay más vértices para visitar, retorna al contexto anterior, produciendo el mismo efecto que el desapilado.

Acción	Motivo	Contenido de la pila
Push P	Inicialización	P
Push R	prox.ciudad no visitada	P R
Push X	prox.ciudad no visitada	P R X
Pop X	no hay mas adyacentes sin visitar	P R
Pop R	no hay mas adyacentes sin visitar	P
Push W	prox.ciudad no visitada	P W
Push S	prox.ciudad no visitada	P W S
Push T	prox.ciudad no visitada	P W S T
Pop T	no hay mas adyacentes sin visitar	P W S
Pop S	no hay mas adyacentes sin visitar	P W
Push Y	prox.ciudad no visitada	P W Y
Push Z	prox.ciudad no visitada	P W Y Z

Tabla 9.1

Conclusiones

Los árboles son una estructura de datos muy utilizada para representar relaciones jerárquicas entre sus elementos, los datos pueden ser accedidos y procesados de manera más rápida que en otras estructuras.

Se ha introducido el tipo grafo por ser una estructura de datos muy utilizada en simulación de distintos tipos de procesos que posee una representación no lineal sencilla que permite combinar las estructuras provistas por el lenguaje Pascal y crear un nuevo objeto con un conjunto de operaciones bien definidas.

Nótese que un grafo generaliza el tipo de datos árbol binario visto anteriormente: hay múltiples relaciones de precedencia y sucesión, que pueden tener un costo.

La idea de tomar una pila para recorrer el grafo es un nuevo ejemplo de utilización de la recursividad y la manera en que se apilan los distintos registros de activación permitiendo a este tipo de algoritmos recordar su contexto anterior.

El lector debe comprender que la recursividad no es una técnica alternativa, sino una manera de programar mejor cierto tipo de algoritmos. La idea es utilizar la definición del problema llevándolo a una versión más simple del mismo tipo.

Ejercicios

1 Se desea ordenar en la memoria una secuencia de 1.500 nombres de ciudades que se leen de teclado.

a) Defina una estructura de datos estática que permita guardar la información leída y calcule su tamaño de memoria.

b) Dado que Pascal no permite manejar estructuras de datos estáticas que superen los 64 KB, se piensa utilizar un vector de punteros a palabras, como se indica en la siguiente estructura:

```
Type      Nombre = String[50];
          Puntero = ^Nombre;
Var Punteros : array[1..1500] of Puntero;
```

b.1) Indique cuál es el tamaño de la variable Punteros al comenzar al programa.

b.2) Escriba el segmento de programa que le permita reservar memoria para los 1.500 nombres. ¿Cuál es el tamaño de la variable Punteros ahora?

b.3) Escriba un módulo para leer los nombres y colocarlos en la variable Punteros.

b.4) Escriba un módulo que permita ordenar la estructura alfabéticamente.

2. Se cuenta con una lista que contiene la información de las ventas realizadas por una empresa de venta de repuestos. Cada venta está compuesta por un código de repuesto y la cantidad vendida. La lista puede contener 0, 1 o más registros por cada código de repuesto, y está ordenada por este campo. Además, se dispone de una estructura auxiliar donde figura el nombre y precio unitario de cada repuesto. Se pide:

   a) Definir las estructuras de datos necesarias para resolver el problema.

   b) Generar una lista de registros conteniendo por cada producto su código, el total vendido y su importe.

   c) Producir un listado con los nombres de los 10 repuestos más vendidos (Observación: puede usarse una estructura auxiliar de solo 10 elementos).

   d) Generar una lista de los repuestos cuya cantidad vendida sea menor que 50. La lista debe ir generándose ordenada por precio unitario a medida que se procesan las listas.

3. Se dispone de una lista con los datos personales y las ventas realizadas (día, producto vendido, monto y tipo de venta -contado o crédito-) por vendedores de productos de belleza. Se pide:

   a) La definición de las estructuras de datos.

   b) La generación de la lista vendedores sin ventas realizadas (ordenada por código de vendedor).

   c) Agregar nuevas ventas.

   d) El proceso que permita imprimir el nombre del vendedor y el total facturado por el mismo (distinguiendo las ventas al contado de las realizadas a crédito).

   e) Modificar el inciso c) para tener discriminadas las ventas de forma tal que primero figuren las realizadas al contado y luego las ventas a crédito.

4. Diseñe un algoritmo para eliminar la primera entrada de una pila.
5. Diseñe un algoritmo que invierta el orden de una lista enlazada.
6. Describa una estructura de datos adecuada para representar la configuración de un tablero durante un juego de ajedrez.
7. Describa una estructura de árbol con la cual se pueda almacenar la historia genealógica de una familia. ¿Qué operaciones se realizan con dicha el árbol?
6. Se tiene un grafo que representa las rutas que conectan diferentes ciudades. Realizar un procedimiento que reciba dicha estructura y determine si existe un camino entre dos ciudades dadas. Determinar el camino que tenga menor costo (mínimo costo equivale a menor distancia).

10

Capítulo 10

Introducción a tipos abstractos de datos

10

Introducción a tipos abstractos de datos

Objetivo

En este capítulo se busca extender el concepto de tipo de dato definido por el usuario como una caracterización de elementos del mundo real, tendiendo al encapsulamiento de la representación y al comportamiento dentro un tipo abstracto de datos (TAD).

Se profundiza en el concepto de división de un programa en módulos. Cada módulo posee dos partes: una visible, donde se especifican las operaciones, llamada interfaz y otra "oculta", denominada implementación o cuerpo del módulo. Esta última debe permanecer inaccesible, pues allí se desarrollan los algoritmos correspondientes a las operaciones antedichas.

Se enfatiza la importancia de la noción de abstracción de los datos y de operaciones para lograr la reutilización del software.

En este contexto, dadas las limitaciones del lenguaje Pascal, se trabaja en el capítulo con una notación correspondiente al lenguaje Ada. El lector notará gran similitud entre las sintaxis de ambos lenguajes. Algunos constructores, propios de Ada se explicarán en el contexto en que se necesiten.

10.1 Abstracciones de datos

Una de las tareas centrales para quienes deseen desarrollar su vida profesional en informática, es la de caracterizar el mundo real para poder resolver problemas concretos mediante el empleo de herramientas informáticas.

Esta "caracterización" significa reconocer los objetos del mundo real y abstraer sus aspectos fundamentales y su comportamiento, de modo de representarlos sobre una computadora.

La utilidad fundamental de la abstracción y la modelización de objetos del mundo real es la posibilidad de reusar soluciones que respondan a dicho modelo en diferentes problemas.

Si se analiza el objeto del mundo real "cliente" de un negocio dado, puede considerarse que tiene: datos personales, atributos que caracterizan su comportamiento y una cuenta corriente. Al mismo tiempo, un cliente efectúa los pedidos, retira y devuelve la mercadería, paga las facturas y puede modificar sus datos básicos y sus atributos.

Esta caracterización rápida sirve para clientes de todo tipo de negocios y, por ende, para una gama muy grande de sistemas informáticos que manejen clientes como objetos del mundo real.

Obviamente, una solución del modelo "cliente" que contempla la estructura de datos que lo representa y la implementación de las operaciones que pueda realizar será reusable en cada uno de estos tipos de sistemas.

En capítulos anteriores se desarrolló el tema Tipo de Datos, empleando tipos estándar para lenguaje dado y la creación de tipos adecuados a un problema determinado; y luego como extensión se intentó crear "bibliotecas de operaciones" para estos nuevos tipos.

En este capítulo, se discute la evolución de esta tarea con el enfoque de una mayor abstracción de los datos.

10.2 Conceptos sobre tipos de datos

Internamente, en la memoria de datos de una computadora no existen "estructuras de datos". Simplemente hay bits en 0 o 1 con una disposición determinada, a partir de una dirección de memoria dada.

Todas las nociones de tipos, estructuras de datos, variables y constantes que se han estudiado son abstracciones para tratar de acercar la especificación de los datos de problemas concretos al mundo real.

Los lenguajes de programación y los sistemas operativos se encargan de manejar naturalmente las conversiones entre el ámbito propio del especialista en Informática y la realidad del hardware de las computadoras.

El concepto de tipo de datos es una necesidad de los lenguajes de programación que conduce a identificar valores y operaciones posibles para variables y expresiones. Un buen principio para el diseño de los algoritmos en un lenguaje de programación es que toda variable debe estar definida de un tipo determinado y toda expresión debe asociarse a un tipo único (para sus operandos internos y su resultado).

Hasta ahora se han considerado tipos de datos simples, estructurados y definidos por el usuario. En lo que sigue se avanzará en el nivel de abstracción, para considerar los Tipos abstractos de datos.

10.2.1 Sistema de tipos

El sistema de tipos de un lenguaje es el conjunto de reglas para asociar tipos a expresiones en el lenguaje.

La mayoría de los lenguajes de programación modernos imponen la obligación de declarar todos los nombres de las variables, asociándoles un tipo. Luego se verifica que las operaciones se hagan entre datos del mismo tipo y la mezcla de tipos normalmente es detectada y considerada un error por los lenguajes de programación, (Sethi, 1992) (Ghezzi, 1987).

A su vez, el sistema de tipos establece reglas, según la operación que se realice con los datos. Por ejemplo, en Pascal si se tiene un operador como `Mod`:

```
resu := dato1 Mod dato2,
```

la regla asociada con `Mod` indica que las variables `dato1`, `dato2` y `resu` deben estar declaradas de tipo entero.

Algunos lenguajes resuelven la mezcla de tipos en expresiones mediante coerción, es decir, una conversión forzada. Por ejemplo, muchos lenguajes admiten que un producto del tipo :

```
resu := dato1 * dato2
```

pueda ser con `resu` de tipo real y que `dato1` y `dato2` puedan tener indistintamente tipo entero o real.

El lenguaje convierte todos los datos a real y realiza el cálculo.

Sin embargo, como principio general, es conveniente que la mezcla de tipos (cualquiera sea) no esté permitida en las expresiones. La coerción debería resolverla explícitamente el programador indicando la conversión a realizar, mediante lo que en lenguajes como C o Ada se denomina conversión explícita o *casting:*

```
resu := REAL(dato1) * REAL(dato2)
```

La verificación de que los tipos usados sean los correctos según la operación en curso es una herramienta muy poderosa para la prevención y corrección de los errores. Se dice que un sistema de tipos es estricto cuando solo acepta expresiones de tipo único.

Los lenguajes que controlan este aspecto tanto en forma estática (en las declaraciones) como dinámica (en ejecución, evaluando el tipo de los resultados intermedios) se llaman fuertemente tipados. Pascal, Modula y Ada son ejemplos de lenguajes fuertemente tipados.

10.3 Módulos, interfaz e implementación

En el capítulo 2 se analizaron las ventajas de descomponer (modularizar) un sistema de software en procedimientos y funciones. Esencialmente se logra abstraer las operaciones, de modo de descomponer funcionalmente un problema complejo.

De aquí en adelante se denominará genéricamente módulo a cada una de las partes funcionalmente independientes en que se divide un programa.

Idealmente un módulo se puede ver como una caja negra con una función interna y una interfaz de vinculación con otros módulos. El resto del programa interactúa con el módulo a través de la interfaz. (Figura 10.1)

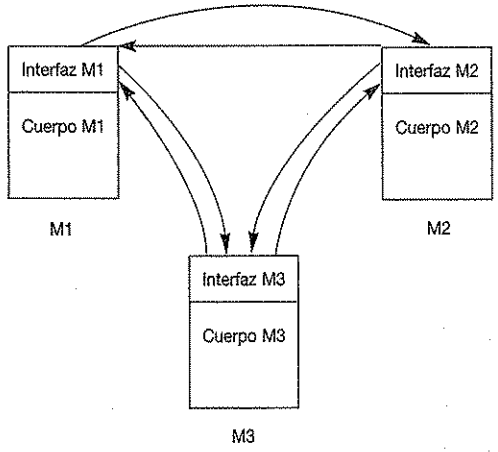


Figura 10.1

La interfaz es una especificación de las funcionalidades del módulo y puede contener las declaraciones de tipos y variables que deban ser conocidas externamente.

La implementación del módulo abarca el código de los procedimientos que concretan las funcionalidades internas.

La interfaz se conoce como la parte pública del módulo, mientras que la implementación es la parte privada.

Ejemplo 10.1

Este programa presenta un pseudocódigo de un tad (sin indicar los parámetros de los procedimientos, para simplificar).

```
MODULO Cliente
   Declaración de la estructura de datos Cliente
   Otras declaraciones públicas.
   Procedure Alta_Cliente;
   Procedure Baja_Cliente;
   Procedure Consultas_Cliente ;
   Procedure Factura_a_Cliente ;

Declaraciones internas del módulo.
   Procedure Alta_Cliente;
      Declaraciones internas del Procedure.
      Cuerpo del procedimiento ;

   Procedure Baja_Cliente;
      Declaraciones internas del Procedure.
      Cuerpo del procedimiento ;

   Procedure Consultas_Cliente ;
      Declaraciones internas del Procedure.
      Cuerpo del procedimiento ;

   Procedure Factura_a_Cliente ;
      Declaraciones internas del Procedure.
      Cuerpo del procedimiento ;
```

10.4 Encapsulamiento de datos

A lo largo del texto se ha reiterado el concepto del desarrollo de especificaciones abstractas que es importante en los sistemas de software. De ese modo, se mejora la calidad de los programas, su legibilidad, su reutilización y su mantenimiento.

Un paso tendiente a la creación de especificaciones abstractas verdaderas o abstracción de datos es lograr **encapsulamiento o empaquetamiento** de los datos: se define un nuevo tipo y se integran en un módulo todas las operaciones que se pueden hacer con él.

Si además el lenguaje permite separar la parte visible (interfaz) de la implementación, se tendrá **ocultamiento de datos** *(data hidding)*.

Y si se logra que la solución del cuerpo del módulo pueda modificar la representación del objeto-dato, sin cambiar la parte visible del módulo, se tendrá independencia de la representación.

Por lo antedicho, se aconsejan aquellos lenguajes que permiten especificar módulos que separan la interfaz de la implementación y que, a la vez, permiten hacer ocultamiento de la información e implementaciones independientes de la representación, para lograr programas reusables y que garanticen un mantenimiento relativamente sencillo.

En el ejemplo anterior, sería razonable que las operaciones sobre un cliente se reflejen en su cuenta corriente. La estructura de datos que representa la cuenta corriente no necesita ser pública. Además, es deseable que las operaciones visibles, tales como la consulta de saldo o el listado de operaciones registradas en la cuenta corriente, sean independientes de la forma interna en que se representa la cuenta corriente del cliente.

Ante un cambio decidido en el curso del mantenimiento del sistema (por ejemplo, el número máximo de operaciones a registrar en la cuenta corriente, o la forma interna de guardar las fechas de las operaciones), es ideal que no se modifique la interfaz que se ha presentado como `Consultas_Cliente`. Más aún, solo el código de la implementación de algún procedimiento interno debería cambiar y esto debería afectar lo menos posible al resto del sistema y mucho menos al usuario.

El tratamiento que se le da en este texto al tema de la abstracción de datos es introductorio y puede ser profundizado por el lector. Conceptualmente es deseable minimizar el número de datos públicos, es decir, declarados en la interfaz de los módulos, y mantener privado el cómo, es decir, la implementación de los procedimientos especificados en la parte pública. Los lenguajes como Pascal estándar tienen limitaciones en este aspecto, por lo que en lo que sigue se tratarán los ejemplos de TAD en Ada.

La sintaxis básica para especificar un módulo en Ada se presenta a continuación.

Ejemplo 10.2

```
Package nombre_modulo is
   < Declaraciones de tipos y subtipos>
   < especificación de procedimientos y funciones>
end nombre_modulo;


Package Body nombre_modulo IS
   < Declaraciones locales, incluyendo inicialización>
   < Implementación de procedimientos y funciones>
   < Instrucciones de inicialización del package de ser necesarias>
END nombre_modulo;
```


10.5 Diferencia entre tipo de dato y tipo abstracto de dato

Un objeto del mundo real es algo más complejo que las representaciones de números o caracteres que se han visto en los tipos de datos simples.

Por ejemplo, al pensar en un banco con sus empleados y los clientes interactuando, se puede ver que hay objetos representables en formas diferentes: empleados, clientes, custodios, cajeros, cajas de seguridad, etc.

Además, estos objetos pueden asociarse de modos diferentes: por ejemplo, no es lo mismo una cola de clientes para cobrar jubilaciones que una cola de clientes para pagar servicios o una cola de clientes para cobrar cheques.

Es decir, no alcanza con pensar en la estructura de registro (tipo) que sirve para representarlos para caracterizar estos objetos: es necesario agregar al tipo las operaciones que caracterizan su comportamiento.

El concepto de **tipo abstracto de dato** es un tipo de dato definido por el programador que incluye:

- Especificación de la representación de los elementos del tipo.
- Especificación de las operaciones permitidas con el tipo.
- Encapsulamiento de todo lo anterior, de manera que el usuario no pueda manipular los datos del objeto, excepto por el uso de las operaciones definidas en el punto anterior.
- La forma de programación utilizada en los capítulos anteriores corresponde a la clásica ecuación de Wirth (Wirth, 1984):

Programa = Datos + Algoritmos

Es decir, dado un problema a resolver, se busca representar los objetos que participan en el mismo e incorporar los algoritmos necesarios para llegar a la solución.

La concepción de este algoritmo se realiza pensando en la solución del problema en su conjunto y en tal sentido se aplican las técnicas de modularización a fin de reducir la complejidad del problema original.

Se debe notar que el comportamiento de los datos no se tiene en cuenta en esta etapa.

De esta forma, el programa implementado se aplica únicamente a la solución del problema original, ya que funciona como un todo. De otra manera, la única relación que existe entre sus módulos tiene como fin la solución de un problema específico.

Sin embargo, esto puede mejorarse si se refina la ecuación anterior. En la ecuación de Wirth la parte Algoritmos se puede expresar como:

Algoritmo = Algoritmo de datos + Algoritmo de control

donde se entiende como **Algoritmo de datos** a la parte del algoritmo encargada de manipular las estructuras de datos del problema, y **Algoritmo de control** a la parte que representa el método de solución del problema, independiente de las estructuras de datos seleccionadas.

Dado que los TAD reúnen en su definición la representación y el comportamiento de los objetos del mundo real se puede escribir la ecuación inicial como:

Programa = Datos + Algoritmos de Datos + Algoritmos de Control

Reemplazando "Datos + Algoritmos de Datos" como TAD se establece la siguiente ecuación:

Programa = TAD + Algoritmos de Control

que describe el enfoque de desarrollo utilizando Tipos Abstractos de Datos.

Se puede notar que en este caso, el programa principal es diferente al que resulta al no utilizar TAD, ya que la parte de Algoritmos de Control se refiere a la relación de los distintos objetos para resolver el problema original, pero el comportamiento de cada objeto se encuentra definido en el TAD.

Para implementar un TAD se requiere especificar un módulo en el que se separan la interfaz (visible) de la implementación (oculta), para lograr el ocultamiento y la protección de los datos.

Es interesante reflexionar que a mayor grado de generalidad en la especificación de las funcionalidades del TAD, se obtienen o se logran mayores posibilidades de reuso. Como se verá más adelante, la esencia de la Ingeniería de Software consiste en incrementar la productividad de los módulos y de los sistemas de software, para lo cual la herramienta fundamental es la posibilidad de reuso de los sistemas que se desarrollen (por este motivo se le da importancia a la especificación de TAD).

A continuación se presentan algunos ejemplos de TAD que se implementan mediante los tipos de datos pilas y colas discutidos en capítulos anteriores. El desarrollo de los programas se efectúa en el lenguaje de programación Ada.

Ejemplo 10.3

El problema consiste en implementar un TAD que permita disponer de un tipo de dato cola, que pueda almacenar números enteros.

```
-- definición del encabezado del Package QueueInt
-- Notese que la definición del tipo Queue es Privada, esto indica
--    que no es necesario que la parte visible del Package incluya
--    la forma del tipo de dato que soporta a la cola
Package QueueInt is
  Type Queue is Private;
  Function Create return Queue;

  Function Push (Q: Queue; Elemento: Integer) return Queue;
  Function Pop (Q: Queue) return Queue;
```



```
  Function Empty (Q: Queue) return Boolean;
  Function Top (Q: Queue) return Integer;
  Function Bottom (Q: Queue) return Integer;
  Private
    Type Cell;
    Type Pointer is access Cell;
    Type Cell is record
      Elem : Integer;
      Sig  : Pointer;
    End record;
    Type Queue is record
      Pri : Pointer;
      Ult : Pointer;
    End record;
End QueueInt;

-- Definición del cuerpo del package, esta parte queda oculta para
--    el usuario del package QueueInt

Package body QueueInt is
  Function Create return Queue is
    Begin
      Return (Null, Null);
    End Create;

  Function Push (Q: Queue; Elemento: Integer) return Queue is
    Aux: Queue := Q;  -- se define una variable y automáticamente
--  se la instancia
    Begin
      If Aux.Ult = Null then
        Aux.Pri := New Cell'(Elemento, Null);
        Aux.Ult := Aux.Pri;
      Else
        Aux.Ult.Sig := New Cell'(Elemento, Null);
        Aux.Ult := Aux.Ult.Sig;
      End if;
      Return Aux;
    End Push;

  Function Pop (Q: Queue) return Queue is
    Aux: Queue := Q;
    Begin
      Aux.Pri := Aux.Pri.Sig;
      Return Aux;
    End Pop;

  Function Empty (Q: Queue) return Boolean is
    Begin
```

```
        Return Q.Pri = Null;
      End Empty;

    Function Top (Q: Queue) return Integer is
      Begin
        Return Q.Pri.Elem;
      End Top;

    Function Bottom (Q: Queue) return Integer is
      Begin
        Return Q.Ult.Elem;
      End Bottom;

  End QueueInt
```

El ejemplo que se plantea a continuación es la implementación de TAD pila. En este caso se permiten "apilar" elementos (sin una cantidad predeterminada *a priori*) y de cualquier tipo.

En este ejemplo se utiliza una característica del lenguaje Ada que permite construir un **TAD genérico**. Se denomina genérico a aquel TAD que no está implementado para un tipo en particular y cada una de las operaciones se resuelve sin tener en cuenta el tipo de dato que se guardará. Posteriormente, se pueden generar instancias de este TAD genérico para algún tipo de dato particular.

Ejemplo 10.4

```
-- El package GenericStack se define genérico, esto es pila de ningún tipo
--    definido a priori
-- Notese que la definición del package comienza con Generic
-- Element es el tipo de dato para cada pila, está definido como privado
--    esto significa que no se implementa el TAD para un elemento
--    particular, para su utilización hay que "instanciar" el package
--    con un tipo de dato particular
Generic
  Type Element is private;
Package GenericStack is
  Type Stack is Private;
  Function Create return Stack;
  Function Push (P: Stack; El: Element) return Stack;
  Function Pop (P: Stack) return Stack;
  Function Empty (P: Stack) return Boolean;
  Function Top (P: Stack) return Element;
  Private

    Type Cell is record
      Elem : Element;
      Sig  : Stack;
```



```
    End record;
  Type Stack is Access Cell;
End GenericStack;

Package body GenericStack is
  Function Create return Stack is
    Begin
      Return Null;
    End Create;

  Function Push (P: Stack; El: Element) return Stack is
    Begin
      Return New Cell'(El, P);
    End;

  Function Pop (P: Stack) return Stack is
    Begin
      Return P.Sig;
    End Pop;

  Function Empty (P : Stack) return Boolean is
    Begin
      Return P = Null;
    End Empty;

  Function Top (P : Stack) return Element is
    Begin
      Return P.Elem;
    End Top;

End GenericStack;
```

Se presentan, a continuación, dos instancias del TAD pila, una para poder almacenar datos enteros y la otra para datos registro.

```
Procedure Usa_TAD_GenericStack;

  Package Pila_de_Enteros is New GenericStack( Integer );

  Type Registro is Record
    nombre_y_apellido is String;
    edad is Integer;
  End Registro;

  Package Pila_de Registro is New GenericStack( Registro );

  ......

End Usa_TAD_GenericStack;
```

El lector debe considerar a partir de estos dos ejemplos la importancia de la utilización de TAD para lograr reusabilidad del código y por tanto una mayor productividad de software.

10.5.1 Ventajas del uso de TAD respecto de la programación convencional

Si bien es factible el desarrollo de algoritmos sin utilizar tipos abstractos de datos, es importante notar el beneficio que tiene su aplicación.

Como se expuso anteriormente, el TAD lleva en sí mismo la representación y el comportamiento de sus objetos y la independencia que este posee del programa o módulo que lo utiliza permite desarrollar y verificar su código de manera aislada. Es decir, es posible implementar y probar el nuevo TAD de una forma totalmente independiente del programa que lo va a utilizar.

Esta independencia facilita la re-usabilidad del código, ya que el nuevo tipo abstracto lleva inmerso su propio comportamiento y no hay efectos colaterales.

Por su parte, el programa o módulo que referencia al TAD, lo utiliza como si fuera una "caja negra" de la que se obtienen los resultados a través de operaciones predefinidas. Esto permite que las modificaciones internas del TAD no afecten a quienes lo utilizan, siempre que su interfaz no se modifique.

Es importante destacar que, mediante el uso de tipos abstractos de datos, es posible para el programador diferenciar dos etapas:

- En el momento de diseñar y desarrollar el TAD no interesa conocer la aplicación que lo utilizará. Esto permite concentrarse únicamente en la implementación del nuevo tipo.
- En el momento de utilizar el TAD no interesa saber cómo funcionará internamente, solo bastará con conocer las operaciones que permiten manejarlo.

10.5.2 Formas de abstracción: tipos de datos abstractos y tipos de datos lógicos

En capítulos anteriores se ha trabajado con pilas y colas. De hecho, el modo en que esto ha sido llevado a cabo constituye un punto intermedio entre tipos definidos por el usuario y tipos abstractos de datos:

1- Se definieron las características del objeto (por ejemplo, pila) sin discutir su implementación.

2- Se definieron las operaciones permitidas sobre el objeto, por ejemplo, `ST_POP` o `ST_PUSH`, indicando como invocarlas, es decir, indicando la interfaz.

En otros términos, al definir pilas y colas se ha realizado abstracción en dos direcciones: abstracción de datos y abstracción de operaciones.

Este tipo de datos se denomina TDL (Tipo de dato lógico) pues existe una conexión lógica, conocida por el usuario entre los tipos, por ejemplo, la declaración del tipo pila, y los procedimientos o funciones asociadas con dicho tipo.

Con los TDL no se llega a un verdadero tipo abstracto de datos porque no existe una vinculación estructural entre dicho tipo y las operaciones asociadas al mismo; es más: los datos declarados dentro del tipo podrían ser accedidos por otras operaciones y, asimismo, las operaciones asociadas con el tipo podrían ser utilizadas para otros datos.

De hecho, no existe lo que se conoce como encapsulamiento del dato abstracto, salvo en la mente del programador o usuario.

Las definiciones conceptuales realizadas para pilas y colas son especificaciones abstractas donde se indica qué operaciones están permitidas y el modo de acceder a ellas. Pero no se ha detallado la representación interna ni se han visualizado los procedimientos en detalle.

Lógicamente, las especificaciones abstractas refuerzan la re-usabilidad y la portabilidad, e incluso independizan al usuario de la implementación de una solución particular dada de la especificación.

10.6 Requerimientos y diseño de un TAD

Disponer de un TAD brinda la posibilidad de tener código reusable, donde se represente no solo la estructura de los datos, sino también su comportamiento. Esto requiere :

- Poder encapsular dentro de un módulo del lenguaje (por ejemplo, el PACKAGE de Ada) tanto la especificación visible como la implementación de las operaciones.
- Poder declarar tipos protegidos (por ejemplo, el PRIVATE de Ada) de modo que la representación interna esté oculta de la parte visible del módulo.
- Poder heredar el TAD, es decir, crear instancias a partir de ese molde, donde se repitan o modifiquen las funcionalidades del tipo, respetando sus características básicas.

El diseño de un tipo de dato abstracto lleva consigo la selección de:

- Una representación interna, lo que implica escoger las estructuras de datos adecuadas para representar la información. Dicha selección está estrechamente relacionada con la complejidad de la implementación de las operaciones.
- Las operaciones a proveer para el nuevo tipo y el grado de parametrización de las mismas.

Estas operaciones pueden clasificarse en:

- Operaciones para crear o inicializar objetos que permiten inicializar la estructura de datos o generar la situación inicial necesaria para crear un objeto nuevo. Por ejemplo:

```
St_Create ( pila )      { genera una pila vacía }
```

- Operaciones para modificar los objetos del TAD que permiten agregar o quitar elementos del TAD. Por ejemplo: `St_Push`, `St_Pop`.
- Operaciones que permiten analizar los elementos del TAD. Por ejemplo: `St_Empty` indica si la pila está vacía.

Si se analiza una posible resolución de problemas de abstracción de datos en Pascal, señalando las limitaciones del lenguaje, el lector observará que no se pueden desarrollar todas las características anteriormente planteadas.

Por otra parte, en las siguientes referencias (USDD ,1980), (Olsen ,1983) y (Habermann ,1983) se puede encontrar un tratamiento detallado de las implementaciones de TAD en lenguaje Ada.

Conclusiones

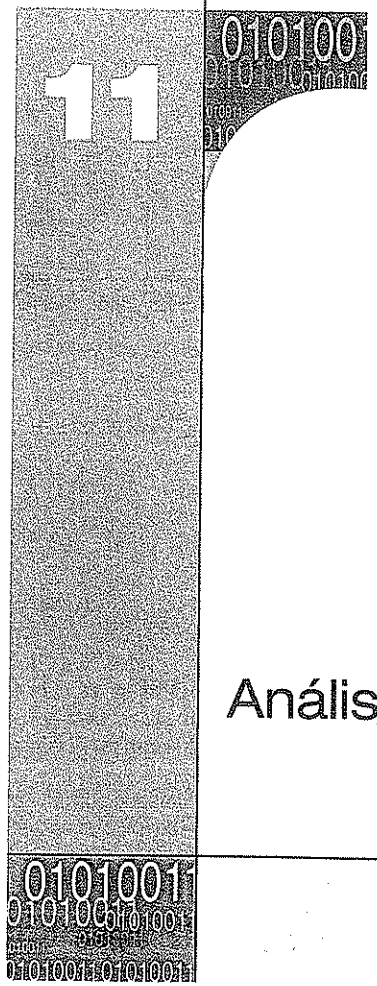
Se ha profundizado la estrategia de abstracción iniciada en los capítulos anteriores.

A la idea de abstracción desde el punto de vista algorítmico, con el objetivo de dividir el problema original en subproblemas de menor complejidad, se agrega la noción de abstracción de los tipos de datos, buscando generalizar las soluciones para incrementar su reusabilidad.

El empleo de TAD conduce al logro de código más fácil de verificar y mantener, aunque esto implique mayor tiempo de desarrollo del software y una posible pérdida de eficiencia en una aplicación particular.

Ejercicios

1. Defina un TAD para manejar números complejos, donde cada número complejo puede representarse por un registro cuyas componentes son dos números reales, uno que representa la parte real y otro la parte imaginaria. Especifique rigurosamente la interfaz del TAD propuesto y luego defina cada una de las operaciones que tendría sentido utilizar sobre este tipo de datos, especifique cómo la implementaría con procedimientos o funciones e indique qué recibiría y qué devolvería con cada una de ellas. Finalmente, implemente las operaciones indicadas previamente.
2. Defina e implemente un TAD para manejar conjunto de palabras, donde cada palabra tenga como máximo 20 caracteres. Indique cuál es la interfaz del TAD. Compare el nuevo tipo con el tipo conjunto provisto por Pascal y finalmente implemente (al menos) la incorporación de palabras al conjunto y la intersección de conjuntos.
3. Defina e implemente los tipos abstractos de datos pilas y colas de un tamaño máximo preestablecido. Indique la representación a utilizar en cada caso y señale las limitaciones que encuentra en la representación propuesta en cuanto al tipo de los elementos de la estructura.



Capítulo 11

Análisis de algoritmos

11

Análisis de algoritmos

Objetivos

En este capítulo se profundizan algunos conceptos relacionados con el desarrollo y la evaluación de los algoritmos, tratando de remarcar que diferentes implementaciones que resuelven correctamente un problema pueden ser muy distintas en cuanto a su eficiencia.

Se generaliza el tratamiento de las ecuaciones de recurrencia en algoritmos recursivos y se muestran las soluciones recursivas e iterativas, así como el proceso gradual de eliminación de la recursividad en el caso de un algoritmo de recorrido de árboles.

Asimismo, se utilizan algoritmos donde intervienen datos de tipo cadena (*string*), los cuales no son numéricos. Por ejemplo, el recorrido de *strings* buscando un *substring* particular, para mostrar diferentes soluciones y el orden de eficiencia de los mismos.

11.1 Repaso a conceptos básicos del análisis de algoritmos

Para la mayoría de los problemas **computables** (es decir, resolubles por una computadora) se pueden tener **diferentes algoritmos** que los resuelven.

El estudio y evaluación de estas alternativas, para poder seleccionar alguna de ellas es un área de investigación importante en Ciencia de la Computación (Sedgewick, 1989).

Se ha visto en el capítulo 8 que la optimización de un algoritmo, en algún sentido, requiere analizar previamente su **eficiencia**, es decir, la utilización que se hace desde el algoritmo de los recursos del sistema donde se ejecuta (básicamente, el tiempo de máquina y la memoria utilizada).

Al estudiar el rendimiento de un algoritmo, también se suele hablar de **complejidad computacional**, tratando de establecer una función matemática que acote el tiempo de ejecución (u otro recurso de interés), **independientemente del conjunto de datos de una corrida específica**. Por ejemplo, se ha visto en el capítulo 8 que el tiempo de ejecución del algoritmo de ordenación *Sort-Merge* está acotado por una función O (N log N).

La eficiencia requiere de un uso mínimo de tiempo de procesamiento y de memoria utilizada. La complejidad computacional es un concepto más elaborado que la eficiencia de un algoritmo, incluso el enfoque de la complejidad de **los sistemas de software**, en base a métricas propias de la Ingeniería de Software es diferente del de la complejidad computacional, asociada con la implementación de algoritmos **sobre un modelo de arquitectura específico**. El tratamiento general del tema excede el alcance de este texto y el lector interesado puede consultar, por ejemplo, (Leeuwen 1990).

11.1.1 Clasificación de algoritmos

Como se mencionó con anterioridad, el parámetro primario para analizar la mayoría de los algoritmos es el **número de datos a ser procesados:** N. Este número puede tener que ver con la cantidad de elementos de un vector, con el número de nodos de un grafo, con los registros de un archivo, etc. En todos los casos interesará relacionar el tiempo de ejecución de un dado algoritmo, **en función de N.**

Se puede tener una primera clasificación de los algoritmos, de acuerdo a su tiempo de ejecución tal como la que sigue:

- **Constante**: se tiene un algoritmo donde cada instrucción se ejecuta una vez o un número fijo de veces, independientemente de los datos, el tiempo de ejecución será constante. Esta situación no es habitual en los casos reales.

- **Log N**: Cuando el tiempo de ejecución del algoritmo crece en forma logarítmica con la cantidad de datos. Es el caso de algoritmos que dividen el espacio de datos del problema en subespacios de menor magnitud, disminuyendo el tamaño del problema original a una fracción constante del mismo (por ejemplo, la búsqueda dicotómica).

- **N**: Cuando hay un pequeño número de acciones de procesamiento sobre cada dato (suponiendo, por ejemplo, su lectura, su multiplicación por dos e impresión), y el tiempo de ejecución crece **linealmente** con N.

- **N Log N**: Es el caso de algoritmos que dividen el espacio de datos del problema en subespacios de menor tamaño, y luego requieren **combinar resultados parciales**. Por ejemplo, un algoritmo de *Sort-Merge* o un procesamiento sobre los datos de una imagen que se tratan por sectores y luego se unen para volver a formar la imagen, tienen este perfil de tiempo de ejecución.

- **N^2**: Cuando el tiempo de ejecución del algoritmo crece en forma cuadrática (como, por ejemplo, en las soluciones elementales de la ordenación de vectores), normalmente significa que los datos son tratados de a pares en un doble lazo repetitivo. El crecimiento del tiempo de ejecución con N pueden ser un limitante para la implementación real con grandes volúmenes de datos.

- **N^3 o 2^N**: Cuando el tiempo de ejecución del algoritmo crece en forma cúbica o exponencial, se trata de soluciones que insumen mucho tiempo y que son difícilmente aplicables usando un número importante de datos. En lo que sigue se verá que a veces las soluciones **directas o de "fuerza bruta"** conducen a tiempos de ejecución que siguen alguna de estas leyes, y esto las hace inaplicables. En estos casos es importante poder **optimizar** el algoritmo **reduciendo la complejidad computacional**, es decir, cambiando la ley cúbica o exponencial por una solución lineal o lineal-logarítmica.

11.1.2 Análisis de caso promedio y caso peor

Al estudiar el rendimiento de un algoritmo, normalmente es de interés conocer su tiempo de respuesta **promedio**. Como se ha analizado en los algoritmos de ordenación utilizando un método heurístico, obtener el número promedio de veces que se ejecuta cada instrucción (y, a partir de ello, el tiempo medio de ejecución) requiere realizar un número importante de ejecuciones del mismo algoritmo con datos generados al azar.

Este enfoque **experimental** tiene varias dificultades: si lo que se mide es **tiempo** hay una dependencia directa de la máquina en la que se ejecuta el algoritmo; si lo que se cuenta es el número de veces que se ejecuta cada instrucción relevante, requiere un volumen de software y/o hardware adicional para poder contabilizarlo; por último, es claramente dependiente de los datos de entrada y muchas veces la función estadística que representa el fenómeno de aparición de los datos no es fácil de determinar (por ejemplo, si se evalúa un algoritmo para la atención de los llamados en una central telefónica, los arribos **no** siguen una ley Gaussiana constante, ni tampoco una curva de Gauss).

Si se desea aplicar un enfoque **teórico**, es decir, estimando o acotando con una función matemática el número de instrucciones a ejecutar, es difícil hacerlo para obtener el caso promedio. En general, es más simple y más fácil determinar el **caso mejor** *y* **el caso peor**.

Desde el punto de vista del estudio de problemas reales, siempre tendrá mayor relevancia el caso peor que el caso mejor. Si se piensa en un software de atención de la central telefónica mencionada anteriormente: no interesa la situación de **ninguna** demanda o de demanda mínima, sino el caso de máxima demanda de acuerdo a los requerimientos. Análogamente, para un algoritmo de ordenación de datos es de interés acotar su comportamiento en el **caso peor** (máximo desorden de los datos) antes que en el caso mejor, en el que los datos han sido previamente ordenados.

Por todo lo expuesto anteriormente, en lo que sigue se analizará la complejidad computacional, tratando de obtener estimadores matemáticos que sean un límite asintótico **máximo** al que puede tender el tiempo de ejecución de los algoritmos **en el caso peor**.

La noción de límite **asintótico** es la que se ha introducido en el capítulo 8.

Definición

El tiempo de ejecución T(n) de un algoritmo se dice "de orden f(n)" cuando existe una función matemática f(n) que acota a T(n)
$T(n) = O(f(n))$ si existen constantes c y n0 tales que $T(n) \leq c\, f(n)$ cuando $n \geq n0$

11.2 Relaciones básicas de recurrencia

En este punto es interesante volver por un momento a los conceptos de recursividad, que se han analizado en el capítulo 7, ya que se discutirá a continuación la expresión de la **relación de recurrencia** que caracteriza a diferentes algoritmos recursivos, para determinar la función matemática que acota su complejidad computacional.

11.2.1 Caso 1: Ley CN = CN-1 + N

El algoritmo recursivo sigue una ley $C_N = C_{N-1} + N$ con $N >= 2$ y $C_1 = 1$

Este es el caso del cálculo del Factorial de N, tal como se vio en el capítulo 7, donde cada invocación recursiva vuelve un paso atrás hasta llegar a la constante C1.

Si desarrollamos

$$
\begin{aligned}
C_N &= C_{N-1} + N \\
&= C_{N-2} + N-1 + N \\
&= C_{N-3} + N-2 + N-1 + N \\
&\ldots\ldots \\
&= C1 + 2 + 3 + 4 + \ldots + N-1 + N \\
&= 1 + 2 + 3 + 4 + \ldots + N-1 + N \\
&= N\,(N+1) / 2
\end{aligned}
$$

Se llega a que el tiempo de ejecución es del orden de $N^2/2$.

11.2.2 Caso 2: Ley CN = CN/2 + 1

El algoritmo recursivo sigue una ley $\mathbf{C_N = C_{N/2} + 1}$ con $N >= 2$ y $C_1 = 0$.

Típicamente, es el caso del cálculo en problemas que dividen el espacio de datos a la mitad en cada invocación (por ejemplo, la búsqueda dicotómica).

Para simplificar el análisis se supone $N = 2^n$, de modo que las divisiones por 2 sean simples porque disminuyen en 1 la potencia de 2 del espacio de los datos.

Por ejemplo $N=64= 2^6$ $N/2=32= 2^5$

Si se desarrolla:

$$
\begin{aligned}
C_N = C_{2^n} &= C_{2^{n-1}} + 1 \\
&= C_{2^{n-2}} + 1 + 1 \\
&= C_{2^{n-3}} + 1 + 1 + 1 \\
&\ldots\ldots\ldots \\
&= C_{2^0} + n
\end{aligned}
$$

Se llega a que el tiempo de ejecución es del orden de n = Log2 N

11.2.3 Caso 3: Ley CN = CN/2 + N

El algoritmo recursivo sigue una ley $C_N = C_{N/2} + N$ con $N >= 2$ y $C_1 = 0$.

Este es el caso del cálculo en problemas que dividen el espacio de datos a la mitad en cada invocación, pero deben examinar todos los datos de entrada.

Para simplificar el análisis nuevamente se supone $N = 2^n$, lo cual no quita generalidad porque siempre existe un n tal que $2^n >= N$ y vale la aproximación indicada.

Si se desarrolla:

$$
\begin{aligned}
C_N = C_{2^n} &= C_{2^{n-1}} + N \\
&= C_{2^{n-2}} + N/2 + N \\
&= C_{2^{n-3}} + N/4 + N/2 + N \\
&\ldots\ldots\ldots \\
&= C_{2^0} + N + N/2 + N/4 + N/8 + N/16 \ldots\ldots
\end{aligned}
$$

Se llega a que el tiempo de ejecución es del orden de $\mathbf{C_N = 2N}$

11.2.4 Caso 4: Ley CN = 2CN/2 + N

El algoritmo recursivo sigue una ley $C_N = 2C_{N/2} + N$ con $N >= 2$ y $C_1 = 0$.

Típicamente, es el caso del cálculo en problemas que dividen el espacio de datos a la mitad en cada invocación, pero deben procesar todos los datos al principio o al final, por ejem-

plo, el *Sort-Merge* visto en el capítulo 8. Se trata de una relación de recurrencia muy utilizada en la solución de algoritmos. Nuevamente se supone $N = 2^n$.

Si se desarrolla: $C_N = C_{2^n} = 2C_{2^{n-1}} + N$

$C_{2^n}/2^n = C_{2^{n-1}}/2^{n-1} + 1$

$= C_{2^{n-2}}/2^{n-2} + 1 +$

$= C_{2^{n-3}}/2^{n-3} + 1 + 1 + 1$

.....

$= C_{2^0}/1 + n$

Se llega a que el tiempo de ejecución es del orden de $C_N = Nn = N \text{ Log } N$

11.2.5 Caso 5: Ley CN = 2CN/2 + 1

El algoritmo recursivo sigue una ley $C_N = 2C_{N/2} + 1$ con $N >= 2$ y $C_1 = 0$.

Este es el caso del cálculo en problemas que dividen el espacio de datos a la mitad en cada invocación tal como el de construcción de la regla marcada que se verá en los puntos siguientes.

Si se desarrolla: $C_N = C_{2^n} = 2C_{2^{n-1}} + 1$

con los mismos métodos anteriores, se llega a que el tiempo de ejecución es del orden de $C_N = 2N$.

Se pueden encontrar en algoritmos recursivos específicos variantes menores de estos 5 casos. De todos modos, la mayoría de los casos que se han tratado en el texto, y que se analizan en lo que sigue de este capítulo, tienen alguna de estas 5 fórmulas de recurrencia y de complejidad algorítmica.

11.3 Soluciones recursivas y no recursivas

En el capítulo 8 se han discutido las ventajas y desventajas de la recursividad, desde el punto de vista de la eficiencia. Se indicó que:

El verdadero uso de la recursividad es el de una herramienta para resolver problemas para los cuales no hay una solución iterativa sencilla. En estos casos la recursividad permite obtener una expresión sencilla del algoritmo en base a la definición del problema original.

A continuación se discuten tres ejemplos de implementaciones donde existen diferencias entre las soluciones recursivas y no recursivas, para que el lector pueda analizar la complejidad computacional para cada caso.

11.3.1 Caso 1: La serie de Fibonacci

La serie de Fibonacci (Pipes, 1958) es una función matemática donde cada término resulta de la suma de los dos anteriores:

$$F_N = F_{N-1} + F_{N-2} \quad \text{siendo } N >= 2 \text{ y } F_0 = F_1 = 1$$

1 1 2 3 5 8 13 21 34 55 89 144 233.....

El desarrollo de una solución recursiva para obtener el N-ésimo término de la serie resulta fácil de implementar:

Ejemplo 11.1

```
function  fibonacci (n: integer) : integer;
  begin
     if  n <= 1  then fibonacci := 1
                 else fibonacci := fibonacci (n-1) + fibonacci (n-2);
  end ;
```

Suponiendo que se quiere calcular Fibonacci(N) con N=12. ¿Cuántas invocaciones a la función se tendrán?

12 ---> 11 y 10
---> 10 , 9 , 9, 8
---> 9 , 8 , 8 , 7, 8 , 7, 7 , 6
..... 144 invocaciones

Puede demostrarse que para calcular Fibonacci(N) con este algoritmo se necesita invocar el procedimiento exactamente el número de la serie de Fibonacci correspondiente a N (en este caso N=12, 144 invocaciones).

En general, como los términos de la serie de Fibonacci siguen una ley d^N donde d=1.61, **el crecimiento de la complejidad computacional de la solución es exponencial con N.**

El error en la elección del algoritmo anterior para resolver la serie de Fibonacci está en el número de veces que se invoca **la misma función.** Resulta claro que si se almacenan los términos bajos a medida que se calculan, se reduce notoriamente el tiempo.

Esto sugiere la conveniencia de una solución **no recursiva:**

Ejemplo 11.2

```
function  fibonacci (n: integer ) :integer;
    const max = 100;  { a lo sumo se calculan 100 términos}
    var i : integer;
        fibo : array [0..max] of integer;
        n: max

begin
      fibo [0] := 1;
      fibo [1] := 1;
      if n>1 then
             for i := 2 to n  do
             fibo [i] := fibo [i - 1] + fibo [i - 2 ];
      fibonacci := fibo [n];
end;
```

Se ha llegado a la misma solución en tiempo **lineal** con N.

Ahora el lector está en condiciones de analizar los tiempos de ejecución de ambos algoritmos para `N` creciente (la sugerencia es reemplazar el tipo `Integer` por `Real`) sobre su computadora.

11.3.2 Caso 2: La construcción de una regla graduada

Muchas de las soluciones recursivas se basan en un lazo de control donde **hay dos llamadas recursivas, cada una de ellas operan sobre la mitad de los datos.** Esta técnica de "divide y vencerás" es muy habitual en el desarrollo de algoritmos y normalmente conduce a un incremento de la eficiencia al reducir el problema global (Meyer, 1978), (Wirth, 1971).

A continuación se analiza la construcción de una regla graduada (supongamos, para simplificar con resolución $1/2^N$) de modo que la marca central tenga una longitud N y las marcas extremas menores tengan una longitud 1, tal como se ve en la Figura 11.1.

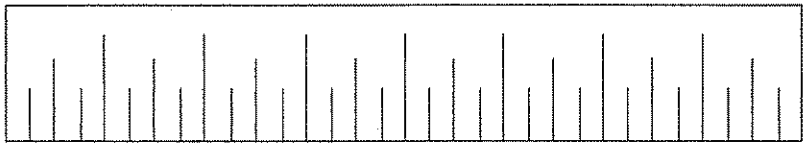


Figura 11.1

La idea de la solución recursiva es obtener un procedimiento que efectúa una marca de altura h (máximo h = n, mínimo h = 1) en una posición x. Naturalmente x irá moviéndose desde el punto medio de la regla (altura máxima) hacia la mitad izquierda y luego hacia la mitad derecha, exactamente a modo de una búsqueda dicotómica, **con la diferencia de que todas las submitades de datos deben tratarse.**

A continuación se muestra el código, la evolución de las invocaciones y el siguiente ejemplo y la Figura 11.2 para un caso concreto.

Ejemplo 11.3

```
procedure regla  ( ini, reso, alto: integer );
  var m: integer;
  begin
      if alto > 0 then
                  begin
                        m:= (ini + reso ) div 2;
                        { pone la marca en el punto medio}
                        marca (m, alto)
                        regla (ini,m, alto - 1);
                        { repite para la submitad izquierda}
                        regla (m, reso, alto -1); {repite para la
                        submitad derecha}
                  end;
  end;
```

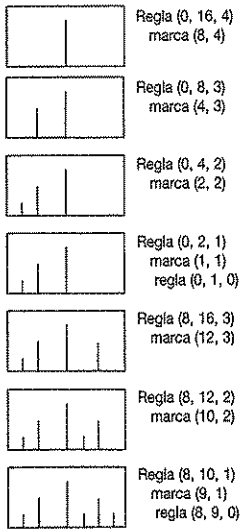


Figura 11.2

Ahora se analiza una solución iterativa que construye la misma regla, **de una forma totalmente diferente**: en cada pasada iterativa va realizando las marcas de cada altura desde h=1=altura mínima hasta h= n=altura máxima.

En la Figura 11.4 y en la Figura 11.3, se ve la solución.

Ejemplo 11.4

```
procedure regla ( ini, reso, alto: integer );
  var x, i, j : INTEGER;
  begin
    j := 1;
    for i:= 1 to alto do
     begin
          for x:= 0 to (ini+reso)div j do
          marca(ini+j+x*(j+j),i); {pone marca en el punto}
          j := j * 2;
     end;
  end;
```

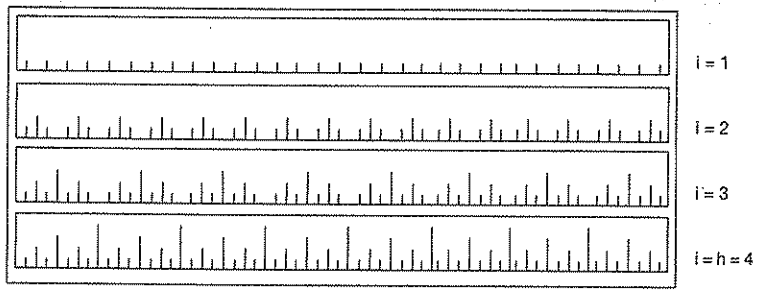


Figura 11.3

Llegado a este punto es interesante que el lector pueda comparar la eficiencia de las dos soluciones, a medida que n crece.

11.3.3 Caso 3: Construcción de un fractal

En algunas ocasiones, la complejidad de la figura geométrica a dibujar o recorrer **marca diferencias a favor de las soluciones recursivas**. Por ejemplo, si se desea elborar un dibujo (Figura 11.4) como el ejemplo siguiente se puede seguir un algoritmo recursivo que se inicia dibujando las 4 puntas y luego el rectángulo interno de **menor a mayor**.

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

Ejemplo 11.5

```
procedure fractal (x, y, r: integer );
 begin
  if  r > 0 then
                begin
                        fractal  (x - r, y  + r, r div 2);
                        fractal  (x +r, y  + r, r div 2);
                        fractal (x - r, y - r, r div 2);
                        fractal (x + r, y -r, r div 2);
                        retangulo (x, y, r);
                end;
 end;
```

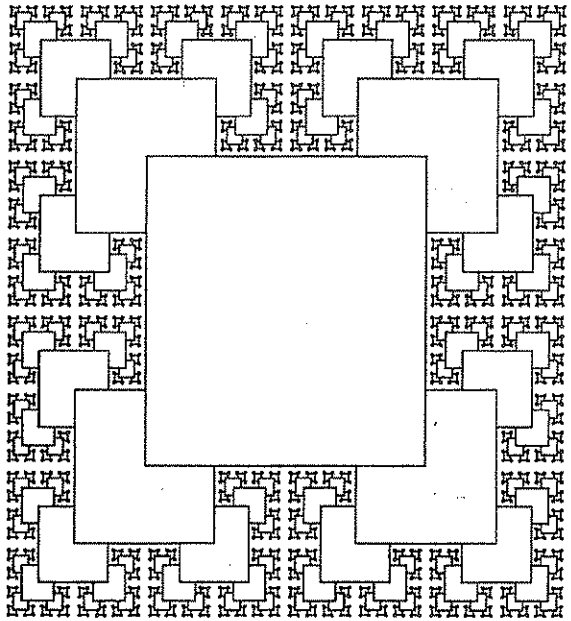


Figura 11.4

Se puede apreciar que resulta más complicado en este caso elaborar una solución iterativa equivalente.

Por otra parte, la memoria y el tiempo de ejecución del algoritmo presentado crece sensiblemente con la dimensión del fractal.

11.3.4 Caso 4: Soluciones recursivas y no-recursivas en el ejemplo de árboles

A continuación se presenta la implementación en Pascal de una biblioteca de procedimientos y funciones para trabajar con árboles binarios. La solución de los mismos puede hacerse en forma recursiva o iterativa por lo que se muestran ambas soluciones.

Luego se define la unidad árbol con la declaración de la estructura y seis operaciones sobre la misma:

Ejemplo 11.6

```
unit arbol;
  interface

    type tipoElem = integer ;
         arbolBinario = ^nodo;
         nodo = record
             elem: tipoElem;
             izq, der: arbolbinario;
         end;
   procedure inicioarbol (var a: arbolbinario);
  {inicializa la estructura  árbolBinario para poder comenzar}

   procedure print (a: arbolBinario);
  {imprime el contenido del árbol}

   procedure delete (var a:arbolBinario, dato:tipoElem);
  {borra todas las ocurrencias de dato en el árbol}

   procedure insert(var a:arbolBinario;dato:tipoElem);
  {inserta dato dentro del árbol de manera que siga ordenado}

   function arbolvacio (a :arbolBinario): boolean;
  {retorna True si el árbol está vacío y False en caso contrario}

   function pertenece (a: arbolBinario;dato: tipoElem): boolean;
  {retorna True si dato es un nodo del árbol}

   implementation
  {aqui va la implementación de los procedimientos y las funciones.
   Elija Ud. si quiere resolverlos en forma recursiva ó no}

 end. {unit árbol}
```

Como puede apreciarse, será necesario inicializar la estructura antes de poder utilizarla.

Una vez que todos los procesos estén implementados, se podrá construir un programa como este:

Código

```
program prueba;
  uses arbol
  var a: arbolBinario;
      valor: integer;
  begin
    write  ("Ingrese valor: ");
    readln (valor);
    inicioarbol (a);
    While (valor <> 0)  do
    begin
        If not pertenece (a, valor) then
        insert (a, valor);
        write ("Ingrese valor");
        readln (valor);
  end;

   Writeln  ('contenido del arbol');
   print (a);
   end.
```

A continuación, se presentan las implementaciones de las operaciones:

11.3.4.1 Inicialización del árbol

La primera tarea a realizar antes de referenciar la estructura es inicializarla, esto se realiza mediante el proceso Inicio de la siguiente forma:

Ejemplo 11.7

```
procedure inicioarbol (var a: arbolBinario);
{inicializa la estructura arbolBinario para poder comenzar}

      begin
          a:= nil;
      end;
```

11.3.4.2 Insertando elementos en un árbol binario ordenado

Por una cuestión de simplicidad, el método que se implementó solo trata de preservar el orden dentro del árbol, no se ocupa de minimizar la altura del mismo. Cada nuevo elemento que se incorpora se inserta siempre como una hoja del árbol. A continuación, se muestra cómo se insertan las secuencias: luz, agua, sal y día (Figura 11.4) y agua, sal, luz y día (Figura 11.5) en un árbol vacío. Las dos frases contienen las mismas palabras, pero el orden de inserción da como resultado árboles distintos.

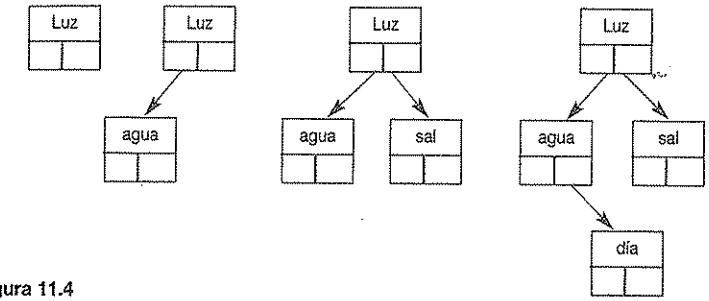


Figura 11.4

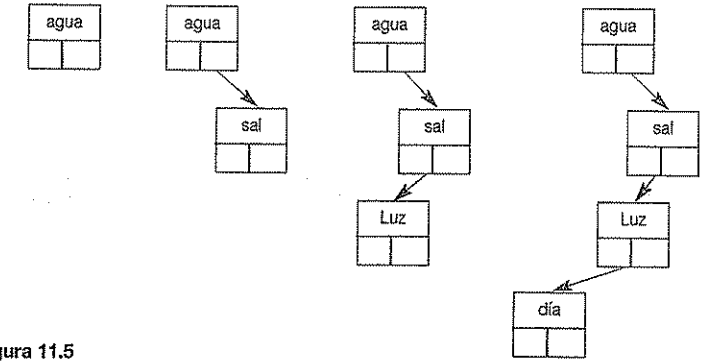


Figura 11.5

El proceso `Insert` busca primero dónde insertar la nueva hoja y luego acomoda los punteros.

Como se puede ver, la versión recursiva es más sencilla que la iterativa:

Ejemplo - versión no recursiva

```
procedure insert (var a: arbolBinario;dato: tipoElem);
  {inserta dato dentro del árbol de manera que siga ordenado}

  var ultimo, actual: arbolBinario;
      fui_a_izq: boolean;
  begin
      ultimo:= nil;
      actual:= a;        {se posiciona al comienzo}
      {mientras no termina la rama, busco la posición para insertar}
      while actual  <> nil  do
      begin
```



```
          ultimo:= actual;
          fui_a_izq:= (dato < actual.elem);
          if fui_a_izq
            then  actual:=  actual^.izq
            else     actual:=  actual^.der;
      end;
{inserta el elemento debajo del  ultimo nodo visitado}
  new (actual;
  actual^.elem:= dato;
  actual^.izq:= nil;
  actual^.der:= nil;
  if  (ultimo = nil)
       then      {el árbol está vacio}
                    a:= actual
       else {se engancha de ultimo según corresponda}
              if  fui_a_izq   then   ultimo^.izq:=  actual
                              else ultimo^.der:=  actual
 end;
```

Ejemplo - versión recursiva

```
procedure insert (var a: arbolBinario;dato: tipoElem);
{inserta dato dentro del árbol de manera que siga ordenado}

  begin
    if a = nil
      then
          begin  {se llegó al final de la rama}
     (1)    new (a);   {se enganchó automáticamente con el padre}
            a^.elem:= dato;
            a^.izq:= nil;
            a^.der:= nil;
          end
      else  if dato < a^.elem  then insert (a^.izq ,dato)
                               else insert (a^.der, dato);

  end;
```

Nótese que la versión recursiva no realiza ningún enganche entre el nivel anterior y la nueva hoja, esto ocurre automáticamente cuando se realiza `New(a)(1)`, siendo un parámetro por referencia que representa la dirección donde se hallará el hijo derecho o izquierdo según corresponda.

Esto es bastante más complicado en la versión iterativa, ya que no solo debe utilizar una variable auxiliar para conservar la posición del padre de la nueva hoja (Ultimo) sino que, además, debe guardar en una variable *boolean* la dirección de selección, para poder saber, de esta forma, si la nueva hoja es un hijo derecho o izquierdo.

Ambas versiones no verifican si el elemento pertenece al árbol o no, con lo cual pueden incluirse elementos repetidos.

11.3.4.3 Visualizando el contenido de un árbol binario ordenado

El objetivo es mostrar los valores del árbol de manera ordenada. En este caso, la versión recursiva es más natural y fácil de escribir que la iterativa.

Para obtener la lista ordenada de los nodos deben listarse para cada uno de ellos, primero los valores que se encuentran en el subárbol izquierdo, luego la raíz y luego los del subárbol derecho (por definición de árbol binario ordenado).

La versión recursiva es más fácil porque se aplica directamente al subárbol correspondiente, no al árbol completo, como se muestra a continuación:

Ejemplo 11.8

```
procedure print (a: arbolBinario);
{imprime el contenido del árbol}

  begin
       if a <> nil    then
          begin
            print (a^.izq);
            write (a^.elem);
            print (a^.der);
          end;
  end;
```

El proceso iterativo es más complicado porque debe "recordar" el estado anterior explícitamente, cosa que no ocurre en la versión recursiva donde cada invocación "apila" automáticamente el entorno anterior.

La idea, entonces, para el proceso iterativo, es continuar por la rama izquierda hasta llegar al final, guardando todos los nodos visitados en una estructura que se comporta como una pila (de manera de volver a verlos en el orden inverso al cual fueron ingresados). Una vez que se imprimió un nodo, se retoma el realizado, se imprime el padre y se continúa con el subárbol encabezado (raíz) por el nodo derecho.

En la siguiente implementación se desprende la potencia de la recursividad:

Ejemplo 11.9

```
procedure print  ( a: arbolBinario) ;
{imprime el contenido del árbol}
  const alturaMax = 100;
  var pila : array[1..alturaMax] of arbolBinario;
      altura: 0..alturaMax;
      subarbol: arbolBinario;
  begin
```



```
      altura:= 0;
      subarbol:= a;
      while (subarbol <> nil) or (altura > 0) do
      begin
        {Se tiene que imprimir los elementos del "SubArbol"
         seguidos por los} {elementos de cada subraíz más los
         elementos que estan en los subárboles izquierdos}
           while (subarbol <> nil) do
           begin
             altura := altura + 1;
             pila[altura] := subarbol;
             subarbol := subarbol^.izq;
           end;
           subarbol := pila[altura]; {se desapila el más chico}
           altura:= altura - 1;
           write (subarbol^.elem, ' ');
           subarbol := subarbol^.der;
      end;
  end;
```

11.3.4.4 Búsqueda en un árbol binario ordenado

Para hallar un elemento determinado se debe utilizar el orden que el árbol posee.

La solución recursiva surge entonces de una manera natural: se compara el valor a buscar con la raíz y si no coincide, continúa la búsqueda por el subárbol derecho (en caso de que sea mayor) o lo hace por el izquierdo (en caso que sea menor). El proceso termina cuando se halla el nodo o cuando se termina el árbol.

Ejemplo 11.10

```
function pertenece  ( a: arbolBinario, dato: tipoelem): boolean ;
{retorna true si dato es un nodo del  rbol }
  begin
    if a = nil
      then pertenece := false
      else if dato = a^.elem
                then pertenece := true
                else if dato < a^.elem
                        then pertenece := pertenece ( a^.izq, dato)
                        else pertenece :=  pertenece (a^,der, dato);
  end;
```

En el caso iterativo, se debe bajar por la rama correspondiente buscando el elemento. A partir de la raíz, se compara el dato a buscar con los nodos del árbol, hasta que se lo encuentre o hasta que se termine la rama.

Ejemplo 11.11

```
function pertenece  ( a: arbolBinario, dato: tipoelem): boolean ;
{retorna true si dato está en el  árbol }
    var auxi: arbolBinario;
    begin
      auxi := a;
        while (auxi <> nil) and (auxi^.elem <> dato) do
          if dato < auxi^.elem
                    then  auxi:=  auxi^.izq
                    else   auxi:=  auxi^.der;

        pertenece :=  auxi <> nil;
    end;
```

La búsqueda de un árbol binario ordenado es, en general, más rápida que la búsqueda secuencial en un vector o una lista. Esto se debe a que el número máximo de elementos examinados en el árbol es igual a la altura y no a la cantidad de nodos que contiene.

En el mejor caso, cuando todas las ramas del árbol tienen aproximadamente la misma longitud, la altura del árbol puede ser del orden de $\log_2(n+1)$, con lo cual, se realiza la misma cantidad de comparaciones que en una búsqueda binaria.

Sin embargo, si el árbol está formado por una rama, la búsqueda degenera en una búsqueda lineal, de allí que sea importante implementar un método de inserción eficiente que mantenga balanceado el árbol (donde las ramas tienen aproximadamente la misma longitud).

11.3.4.5 Borrando un nodo en un árbol ordenado

El suprimir un nodo de un árbol ordenado es una tarea que se debe realizar con cuidado ya que se pretende que cuando se quite el nodo, el árbol continúe ordenado.

Ciertos nodos son fáciles de remover:

Una hoja se borra directamente cambiando un puntero a `nil`.

- Si un nodo tiene un solo hijo, borrarlo representa poner a su hijo en su lugar.
- Pero si un nodo tiene dos hijos no es tan simple:

  a) Como una primera solución, es posible borrar el nodo, poniendo a su hijo izquierdo en su lugar y luego reordenando el hijo derecho en el lugar apropiado. El problema de esta solución es que incrementa la altura del árbol (cosa esperada solo cuando se inserta un nodo, no cuando se borra).

  b) La mejor solución para borrar un nodo de un árbol es la siguiente: buscar el nodo de mayor valor dentro del subárbol izquierdo del nodo a borrar y sacar este nodo del árbol. Esta es una tarea fácil ya que este nodo tiene a lo sumo un hijo. Finalmente se reemplaza el nodo a borrar por el mayor del subárbol izquierdo. Esto se muestra en la Figura 11.6.

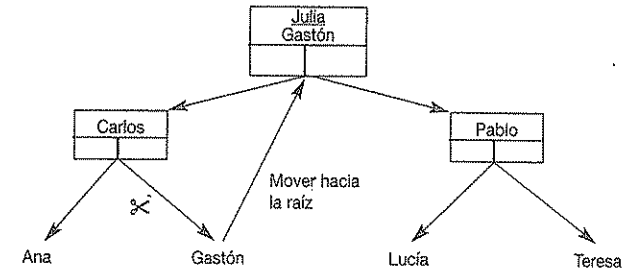


Figura 11.6

El siguiente *procedure* implementa esta solución:

Ejemplo 11.12

```
procedure  delete (var a: arbolBinario; dato:tipoElem);
{Borra todas las ocurrencias de dato en el árbol}

    var nodoviejo : arbolBinario;
    procedure cambioderaiz  (var nodo : arbolBinario);
    { Reemplaza el raíz del árbol  por el nodo que contiene
      el mayor valor del subarbol nodo}
      var maxnodo: arbolBinario;
      begin
        if nodo^.der = nil then
          begin                          {si nodo^.elem es el mayor}
            maxnodo := nodo;             {nodo a mover}
            nodo:= nodo^.izq;            {lo saca del subárbol}
            maxnodo^.der:= a^.der        {se conectan las ramas}
            maxnodo^.izq:= a^.izq;
            a:= maxnodo;                 {se convierte en la raiz}
          end
        else CambioDeRaiz (nodo^.der);    {se busca el máximo}
      end;                          {fin del procedure cambio de raiz}

  begin
    if a <> nil then
      if dato = a^.elem then
        begin
          nodoviejo := a;
          if a^.izq =nil
            then a:=  a^.der
            else if a^.der = nil
                   then  a := a^.izq
                   else cambioderaiz (a^.izq);
```

```
            dispose (nodoviejo);
            delete (a,dato);
        end
        else if dato < a^.elem
                then delete (a^.izq, dato)
                else delete (a^.der,dato);
    end;
```

La versión no recursiva del procedimiento se deja como ejercitación.

11.3.5 Transformación de soluciones recursivas en no recursivas

En los algoritmos anteriores se puede apreciar que **dada una solución recursiva, siempre es posible tener una solución no recursiva del mismo problema.**

Asimismo (tal como se vio en el caso del algoritmo para la serie de Fibonacci) **no siempre una solución recursiva clara y concisa significa tener eficiencia en la ejecución.** Muchas veces es posible construir soluciones algo más confusas de leer, pero más eficientes en tiempo con técnicas no recursivas.

Si bien la recursividad es un concepto fundamental en Ciencias de la Computación, su tratamiento profundo excede el alcance de este texto, pero el lector interesado puede, por ejemplo, consultar (Betley, 197); (Fisher, 1964) para tener información más completa.

11.4 Soluciones de algoritmos con estructuras de datos estáticas o dinámicas

Como se vio en el capítulo 10, la noción de un puntero permite construir y manipular listas encadenadas de diferentes tipos. El concepto de puntero introduce la posibilidad de ensamblar una colección de bloques denominados nodos, en estructuras flexibles. Mediante la alteración de los punteros, los nodos, por ejemplo, se pueden enganchar y desenganchar.

Se establece un conjunto fijo de nodos representados por un arreglo y un puntero a un nodo se representa por la posición relativa del nodo dentro del arreglo. La desventaja de este método es doble. La primera es que el número de elementos que se requiere no puede predecirse en el momento en que se escribe el programa. Generalmente los datos con los cuales se debe ejecutar el programa determina el número de nodos necesarios.

Por consiguiente, no interesa cuántos elementos contiene el arreglo de nodos, ya que siempre es posible que se presente la ejecución del programa con una entrada que requiera un número mayor de nodos.

La segunda desventaja del método del arreglo es que cualquiera que sea el número de nodos declarados, estos permanecerán asignados al programa a través de su ejecución.

Por ejemplo, si se declaran 500 nodos de un tipo determinado, la cantidad de almacenamiento requerido para estos 500 nodos se reserva para este fin. Si el programa solo utiliza 100 o aún 10 nodos de ejecución, los nodos adicionales permanecerán reservados y su almacenamiento no se puede utilizar para algún otro fin diferente a este.

La solución a este problema es tener nodos que sean dinámicos en lugar de nodos estáticos. Es decir, que cuando se requiera un nodo, se reserve el almacenamiento para él, y cuando ya no se necesite, se libere el almacenamiento. Dicho almacenamiento para estos nodos que ya no son utilizados, queda disponible para cualquier otro uso.

Además, no se establece un límite predefinido para la cantidad de nuevos nodos.

Siempre y cuando exista suficiente almacenamiento para el programa, parte del almacenamiento puede ser reservado para ser utilizado como un nodo.

La mayor desventaja de la implementación dinámica es que se puede consumir mucho tiempo llamando al sistema para asignar y liberar el almacenamiento. Su mayor ventaja es que los nodos no son reservados con anticipación. Por ejemplo, si un programa utiliza dos tipos de listas: lista de enteros y lista de caractere. Bajo la representación de arreglo se requeriría asignar en forma inmediata dos arreglos de tamaño fijo. Si un grupo de listas produce desbordamiento en el arreglo, el programa no puede continuar y se deben realizar test adicionales. En el caso de representaciones dinámicas se definen dos tipos de nodos desde el principio, pero no se asigna ningún almacenamiento para las variables hasta que no sean requeridas. A medida que son requeridos los nodos, el sistema se encarga de proporcionar el almacenamiento. Cualquier almacenamiento no utilizado por un tipo de nodo puede ser utilizado por el otro. Por consiguiente, siempre y cuando exista suficiente almacenamiento disponible para los nodos presentes en las listas, no se presentará ningún desbordamiento.

Otra ventaja de la implementación dinámica es que la referencia al puntero no involucra el cálculo de dirección que se requiere en el caso de acceder a un elemento del arreglo. Para calcular la dirección del vector (p) el contenido de p debe ser agregado a la dirección base del arreglo vector, mientras que la dirección p está dada por el contenido de p directamente.

La mayoría de los lenguajes de programación no tienen pilas y colas como tipo de datos estándar. A continuación se muestran algunas de las operaciones con colas de enteros usando estructuras de datos estáticas y dinámicas. Para ejemplificar se han tomado las siguientes operaciones sobre el tipo de dato: vaciar, longitud, poner y sacar elementos.

11.4.1 Colas

11.4.1.1 Implemetación estática

Se define una cola como un arreglo finito de 100 elementos de tipo entero y dos variables de igual tipo que indican el comienzo (`front`) y el final (`rear-1`) de la cola.

Definición de la estructura:

Ejemplo 11.13

```
const  maxqueue = 100;
type queue = record;
            item : array [1..maxqueue] of elemento;
            pri,ult : 1..maxqueue;
       end;
var  q : queue;
     elem : elemento;
begin
     q.pri := maxqueue;
     q.ult :=  maxqueue;
```

A continuación se presenta la implementación de las siguientes operaciones:

- `Q-empty` (nombre-variable-cola)
- `Q-length` (nombre-variable-cola)
- `Q-push` (nombre-variable-cola, elemento)
- `Q-pop` (nombre-variable-cola, elemento)

A continuación las líneas de códigos de las operaciones:

Ejemplo 11.14

```
function q- empty ( q: queue ): boolean;
   begin
      with q do
         if  pri = ult then q-empty := true
                       else q-empty := false;
   end;
```

Ejemplo 11.15

```
function q- length  ( q: queue ): integer ;
   begin
     with q do
       if pri > ult  then  q- lenght := ult-pri;
                     else  q- lenght := pri-ult
   end;
```

Ejemplo 11.16

```
procedure q_push (var q: queue;elem: elemento) ;
  begin
    with q do
    begin
      if ult = maxqueue then ult := 1
                        else ult := ult  +1;
      if ult = pri then  writeln (' error estructura llena')
                   else   item [ult]:= elem;
  end;
```


Ejemplo 11.17

```
procedure q_pop (var a: queue, ele: elemento);
   begin
       if q_empty (q)
           then writeln ( ' error, cola vacía')
           else  with q do
                 begin
                    if pri = maxqueue then pri := 1
                                      else pri := pri + 1;
                    ele:= item[pri ];
                 end;
   end;
```

11.4.1.2 Implementación dinámica

Se define la cola como una lista enlazada de nodos de tipo entero y una variable en la que se guarda el puntero al último elemento agregado para no tener que recorrer la lista cada vez que se quiera agregar un elemento.

Definición de la estructura:

Ejemplo 11.18

```
ptr = ^nodo;
nodo = record
         dato : elemento;
         sig  : ptr;
       end;
queue = record
          totalnodos : Integer;
          pri, ult : ^nodo;
        end;
      var q : queue;
          ele : elemento;
      begin
        q.pri: nil;
        q.ult : nil;
        q.totalnodos := 0;
      end
```

Ejemplo 11.19

```
function  q_empty ( q: queue): boolean;
  begin
    if  pri = nil  then  q_empty := true
                   else  q_empty := false;
  end;
```

Ejemplo 11.20

```
function q_length (q: queue): integer;
  begin
    q_length:= q.totalnodos;
  end;
```

Ejemplo 11.21

```
procedure  q_push  (q: queue, elem: elemento);
    var ptrnodo : ^ nodo;
     begin
       { se da lugar para el nuevo nodo }
       New  (ptrnodo ) ;
       ptrnodo ^.sig:= nil;
       {este, pasa a ser el ultimo de la cola}
       q.ult^.sig := ptrnodo;
       q.ult := ptrnodo;
       {se carga el valor que se debe asignarle al nodo}
       ptrnodo^.dato:=  elem;
       q.totalnodos:=  q.totalnodos + 1;
     end;
```

Ejemplo 11.22

```
procedure q_pop (q:queue, var elem: elemento);
    var ptrnodo: ^nodo;
    begin
      ptrnodo := q.ult;
      elem := q.ult ^.dato;
      q.ult := q.ult^.sig;
      dispose (ptrnodo);
      q.totalnodos:= q.totalnodos - 1;
    end;
```

Se dejan como ejercitación adicional las implementaciones de las operaciones restantes y las operaciones para manejar una pila.

11.5 Análisis de algoritmos no numéricos: tratamiento de strings.

Muchas veces los datos a procesarse no se descomponen lógicamente en piezas más pequeñas. Este tipo de datos se caracteriza solo por el hecho de que pueden escribirse como un ***string***, o sea, una secuencia lineal de caracteres (generalmente larga).

Obviamente los *strings* son importantes en los sistemas de procesamiento de palabras, los cuales poseen diferentes posibilidades para la manipulación de texto. Este tipo de sistema procesa *strings* de texto, al cual se lo define como una secuencia de números, letras y caracteres especiales. Estos objetos pueden ser grandes (por ejemplo, este libro posee más de quinientos mil caracteres), por lo tanto la manipulación de los datos juega un rol importante en la eficiencia de los algoritmos.

Otro tipo de *strings* son los binarios, representados por una secuencia de ceros y unos que son un tipo especial de *string* de texto, y son usados en muchas aplicaciones. Por ejemplo, algunos sistemas gráficos representan las imágenes como *strings* binarios.

Una operación fundamental sobre *strings* es la búsqueda de un patrón (*pattern matching*): dado un *string* de texto de longitud N y un patrón (*pattern*) de longitud M, se desea encontrar una ocurrencia del patrón en el texto. Se utiliza el término texto, tanto para referir una secuencia de ceros ó unos ó cualquier otro tipo especial de *strings*. Muchos algoritmos para este problema pueden extenderse fácilmente para encontrar todas las ocurrencias del patrón en el texto, dado que se puede buscar el texto secuencialmente empezando cada vez desde el punto siguiente al principio de un encuentro (*match*) para encontrar el siguiente:

Este problema se puede caracterizar como un problema de búsqueda, con el patrón como clave. A continuación se presentan algoritmos de búsqueda.

11.5.1 Algoritmo trivial (fuerza bruta, *brute force*)

El método obvio para la búsqueda de un patrón consiste en chequear cada posible posición en el texto en donde el patrón puede encontrarse. El programa siguiente busca de esta forma la ocurrencia del patrón p[1..M] en el *string* de texto a [1..N].

Ejemplo 11.23

```
function brutesearch : integer;
  var i, j: integer;
  begin
    i := 1;
    j := 1;
    repeat
      if a[i] := p[j]  then begin
                              i:= i + 1;
                              j:= j + 1;
                            end
                            else  begin
                              i := i-j+2;
                              j := 1;
                            end;

    until j > M or i > N;
    if j > M  then  brutesearch := i-M
              else  brutesearch := i;
  end;
```

Este programa mantiene un puntero (i) en el texto y otro puntero (j) en el patrón. Mientras los dos caracteres apuntados son iguales, ambos punteros se incrementan. Si se llega a alcanzar el fin del patrón (j > M), entonces se ha encontrado el patrón. Si i y j apuntan a caracteres distintos, se inicia j al principio del patrón e i se le asigna la posición siguiente desde donde se empezó a encontrar caracteres coincidentes. Cuando se alcanza el fin del texto (i > N), entonces no hay coincidencia. Si el patrón no se encuentra en el texto, se retorna el valor N+1.

Este algoritmo puede ser lento para algunos patrones, por ejemplo, si el texto es binario, tal como ocurre en el procesamiento de imágenes. En la Figura 11.7 se presenta un algoritmo que busca el patrón 10100111 en un *string* binario largo.

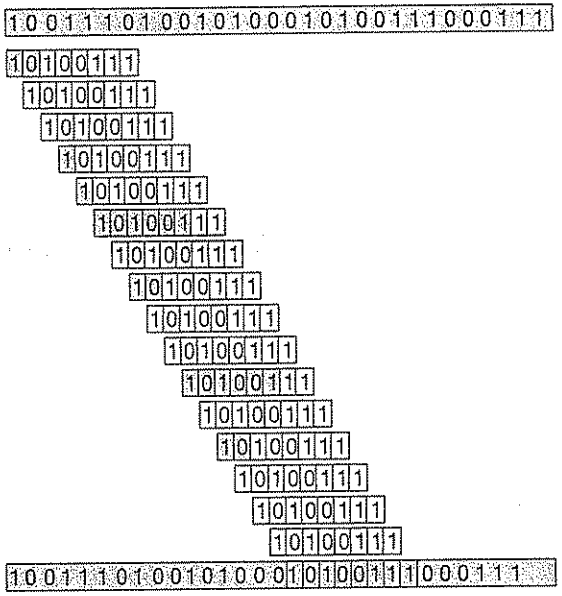


Figura 11.7

Para cada línea excepto la última, en la que se encuentra la coincidencia, consiste de cero o más caracteres que coinciden en el patrón, seguido de un bit que no coincide. Estas son los **comienzos falsos** (***false starts***) que ocurren cuando se trata de encontrar la coincidencia. Un objetivo obvio en el diseño del algoritmo es limitar el número y la longitud de tales comienzos. Para este ejemplo, cerca de dos caracteres son examinados en promedio para cada posición, y esto podría ser mucho peor.

Propiedad: El algoritmo fuerza bruta (*Brute Force*), requiere cerca de N x M comparaciones de caracteres.

El peor de los casos se presenta cuando el patrón y el *string* son todos ceros seguidos de un 1. Entonces, para cada N - M + 1 posibles posiciones de coincidencias, todos los caracteres en el patrón son chequeados en el texto, para tener un costo total de M (N - M + 1). Normalmente M es pequeño comparado con N, por lo cual el costo es cerca de N x M . En la figura 11.7; N = 30, M = 8.

Este *string* particular no ocurre en textos de Castellano, Inglés o Pascal, pero puede ocurrir cuando se opera con textos binarios, por lo tanto se debe disponer de mejores algoritmos.

11.5.2 Algoritmo de Knuth-Morris-Pratt

La idea básica de este algoritmo es la siguiente: cuando no se detecta coincidencia, el contenido de *false start* consiste de caracteres que se conocen de antemano (dado que están en el patrón). De alguna manera se debe tomar ventaja de esta información en lugar de actualizar el puntero (i) sobre todas las posiciones de caracteres conocidos.

Un ejemplo simple de este caso se tiene cuando el primer carácter en el patrón no se repite dentro del patrón (por ejemplo, 10000000). Se supone que se tiene un *false start* (j) caracteres en alguna posición en el texto. Cuando se detecta la no coincidencia, se sabe que j caracteres han coincidido, entonces no se tandrá que actualizar el puntero i, dado que ninguno de los previos j - 1 caracteres en el texto van a coincidir con el primer carácter del patrón. Este cambio se puede hacer reemplazando la instrucción i = i - j + 2 del programa por i = i + 1. El efecto práctico de este cambio es limitado porque este tipo de *string* no es habitual que se presente, pero la idea de este algoritmo sirve para la generalización del mismo en el algoritmo de Knuth-Morris-Pratt. Sorpresivamente, siempre se puede ordenar de manera que el puntero i nunca se decremente.

Por ejemplo, cuando se busca 10100111 en 101010011, primero se detecta que el quinto carácter no coincide, pero se debe continuar en el tercer carácter para continuar la búsqueda, porque de otra manera se pierde dicha coincidencia.

Se utiliza el arreglo `next [1..M]` para saber cuánto se puede retroceder para encontrar otro comienzo de coincidencia. Se puede pensar que se hace una copia de los primeros j - 1 caracteres del patrón, desde la izquierda a la derecha, comenzando con el primer carácter de la copia sobre el segundo y se detiene cuando se sobreponen todos (o no hay más). Si se detecta una falta de coincidencia en P[j], esta superposición de caracteres definirá el próximo lugar posible en donde el patrón podrá coincidir, la distancia que se debe volver en el patrón next[j] es exactamente uno más el número de caracteres superpuestos. Específicamente, para j > 1, el valor de next [j] es el máximo k < j para lo cual los primeros k - 1 caracteres del patrón concomitente con los k - 1 caracteres del los primeros j - 1 caracteres del patrón. Como se observa, es conveniente inicializar `next [1]` en cero.

El arreglo `next` da inmediatamente un camino para limitar (de hecho elimina) el retorno del puntero i, como se ha discutido más arriba. Cuando i y j apuntan a caracteres distintos (evaluando por una coincidencia comenzando de la posición i - j + 1 en el *string* de texto), entonces la posible posición para una coincidencia comienza en `i-next [j] + 1`. Por definición de la ta-

bla next, los primeros next [j]-1 caracteres en esa posición coinciden con los primeros next [j] -1 caracteres del patrón, no se necesita actualizar el puntero i muy lejos, simplemente se deja a i sin cambiar, y se establece j a next [j] como se muestra en el siguiente módulo.

Ejemplo 11.24

```
procedure inicializarnext;
  var i, j: integer;
  begin
    i := 1;
    j := 0;
    next [1] := 0;
    repeat
      if ( j = 0 ) or ( p[i] = p[j] ) then
        begin
          i := i + 1;
          j := j +1;
          next [i] := j;
        end
        else j := next [j];
    until i > = M;
  end;
function busquedakmp : integer;
  var i, j : integer;
  begin
    i := 1;
    j := 1;
    inicializarnext;
    repeat
      if ( j = 0 ) or ( a [i] = p [j] ) then
        begin
          i := i + 1;
          j := j + 1;
        end
        else j:= next [j];
    until ( j > M )  or  ( i >  N );
    if j > M
      then busquedakmp := i - M
      else busquedakmp := i;
  end;
```

Cuando j = 1 y a [i] no coincide con el patrón, entonces no hay superposición, por lo tanto, se debe incrementar i y poner j al principio del patrón. Esto se logra definiendo next [1] en cero, lo que resulta que j se lleve a cero, entonces se incrementa i y j en la próxima iteración.

Es necesario tener en cuenta que en Pascal se debe declarar el arreglo desde 0, evitando la declaración estándar, que dará error *Subscript out of rang* (fuera de límite del rango) cuando j = 0. Funcionalmente este programa es igual al ***brutesearch***, pero es más eficiente cuando los patrones son repetitivos.

Resta aún el cálculo de la tabla next. Básicamente, es el mismo programa que fue presentado anteriormente. Cuando se incrementa i y j, esto determina que los primeros j - 1 caracteres del patrón coincidan con los caracteres en las posiciones p [i - j-1..i -1] los últimos j - 1 caracteres con los primeros i - 1 caracteres del patrón. Este es el j más grande que mantiene la propiedad, dado que de otra manera **una posible coincidencia** del patrón con sí mismo puede perderse. J es el valor asignado a next [i].

Una manera interesante de ver este algoritmo es considerar al patrón como fijo, por lo tanto, la tabla next puede ser construida por el programa. Por ejemplo, el módulo que se presenta a continuación es equivalente al programa anterior, para el patrón que se ha considerado, pero mucho más eficiente.

Ejemplo 11.25

```
i:= 0;
0:  i:= i + 1;
1:  if a [i]  <>  "1"  then goto 0; i := i +1;
2:  if a [i]  <>  "0"  then goto 0; i := i +1;
3:  if a [i]  <>  "1"  then goto 0; i := i +1;
4:  if a [i]  <>  "0"  then goto 0; i := i +1;
5:  if a [i]  <>  "0"  then goto 0; i := i +1;
6:  if a [i]  <>  "1"  then goto 0; i := i +1;
7:  if a [i]  <>  "1"  then goto 0; i := i +1;
8:  if a [i]  <>  "1"  then goto 0; i := i +1;
9:  if a [i]  <>  "1"  then goto 0; i := i +1;
    busqueda : = i - 8;
```

Los rótulos de los goto corresponden precisamente a los de la tabla next. El procedure inicializarnext puede modificarse fácilmente para producir este programa. Para evitar chequear i > N cada vez que se incrementa i, se supone que el patrón está almacenado al final del texto (como un centinela) en a[N +1..N + M].

El programa anterior solo utiliza unas pocas operaciones básicas para resolver el problema de coincidencia. Esto indica que se puede resolver con una máquina simple, de las llamadas máquinas de estado finito. La Figura 11.8 presenta la máquina de estado finito para el programa anterior.

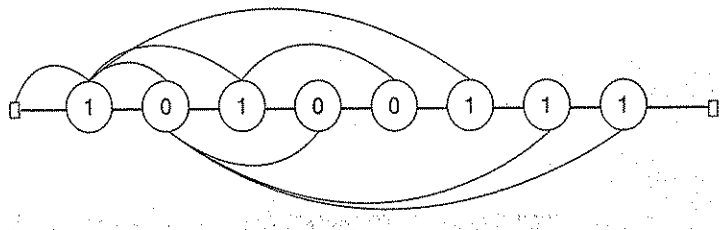


Figura 11.8

Esta máquina consiste de **estados** (indicados por los números encerrados con un círculo) y transiciones (indicadas por líneas). Cada estado tiene dos tipos de transiciones saliendo de él, las primeras son transiciones de coincidencia (línea rectas) y las segundas las transiciones de no coincidencia (líneas curvas). Los estados son transiciones y los estados son `goto`. Cuando está en un estado llamado **x** la máquina puede ejecutar una sola instrucción: **si el carácter actual es x, pasa a la transición de coincidencia, en caso contrario pasa a no coincidencia**. Cuando pasa un carácter, el próximo carácter en el *string* es el carácter corriente. Existen dos excepciones, el primer estado siempre toma una transición de coincidencia y pasa al siguiente carácter (especialmente esto corresponde a buscar la primera ocurrencia del primer carácter del patrón) y el último estado, parar, indica que se ha encontrado una coincidencia.

Se pueden hacer algunas mejoras en este algoritmo, porque no se tiene en cuenta el caracter en el cual se produce la no coincidencia, por ejemplo, para el caso en que el texto comience con 101 y el patrón sea 10100111. Después de la coincidencia 101, se encuentra un caracter que no coincide, en la cuarta posición, en este punto la tabla next indica que se controle el segundo carácter del patrón, en lugar del cuarto carácter del texto, dado que la base de la coincidencia 101, el primer caracter puede estar desplazado con el tercer caracter del texto (pero no se debe comparar esto porque se conoce que ambos son 1). Sin embargo, no existe coincidencia, dado que por la no concordancia se conoce el próximo carácter, el cual no es un cero como lo requiere el patrón. La siguiente Figura 11.9 presenta la versión mejorada de la máquina de estado finito.

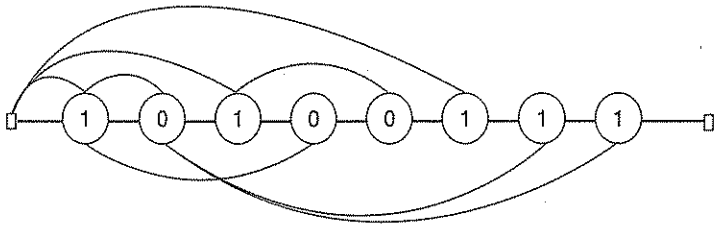


Figura 11.9

Es sencillo llevar a cabo los cambios en el algoritmo. Se debe reemplazar la instrucción `next[i]:= j`, en el algoritmo `inicializarnext` por:

```
If  p[j] <> p[i]
      Then   next [i] := j
      else  next [i] : = next [j]
```

Propiedad: el algoritmo Kunth-Morris-Pratt nunca utiliza más de M + N comparaciones de caracteres.

11.5.3 Algoritmo de Boyer-Moore

Si retroceder no es dificultoso, entonces se puede desarrollar un algoritmo de búsqueda en *strings* que busque el patrón desde derecha a izquierda tratando de localizar coincidencias con el texto. Cuando se busca el patrón 10100111, si se encuentran coincidencias para el octavo, séptimo y sexto caracter, pero no para el quinto, se puede inmediatamente correr el patrón siete posiciones a la derecha y chequear el quinto caracter, dado que la coincidencia parcial 111 puede aparecer en cualquier lugar del patrón.

Una versión de la tabla `next` (del patrón 10110101) de derecha a izquierda se muestra en la Figura 11.10.

j	next[j]	
2	4	1 0 1 1 0 1 0 1
		1 0 1 1 0 1 0 1
3	7	1 0 1 1 0 1 0 1
		1 0 1 1 0 1 0 1
4	2	1 0 1 1 0 1 0 1
		1 0 1 1 0 1 0 1
5	5	1 0 1 1 0 1 0 1
		1 0 1 1 0 1 0 1
6	5	1 0 1 1 0 1 0 1
		1 0 1 1 0 1 0 1
7	5	1 0 1 1 0 1 0 1
		1 0 1 1 0 1 0 1
8	5	1 0 1 1 0 1 0 1
		1 0 1 1 0 1 0 1

Figura 11.10

En el caso presentado en la Figura 11.10, caso `next [j]` es el número de caracteres por el cual el patrón debe correrse a la derecha, dada una no coincidencia buscando desde la derecha, ocurrida en el j-ésimo carácter del patrón, similar a la implementación del algoritmo Knuth-Morris-Pratt. No se analiza esto con más detalles porque es similar a lo visto anteriormente, solo que la comparación es de derecha a izquierda.

El algoritmo del caracter no encontrado es sencillo de implementar. Simplemente mejora el algoritmo `BruteForce` derecha-a-izquierda, se inicializa un arreglo llamado *skip* que indica para cada caracter en el alfabeto cuántos se puede saltear, si ese carácter aparece en el texto y causa una no coincidencia durante la búsqueda en un *string*. Debe existir una entrada en *skip* para cada caracter que posiblemente pueda ocurrir en el texto, por simplicidad, se asume una

función `index` (c: char): `integer`, que retorna cero para blancos e i para la i-ésima letra en el alfabeto, también se supone un `procedure initskip`, el cual inicializa el arreglo *skip* para M caracteres que no están en el patrón y para j desde 1 a M, establece `skip[index(p[j])]` a M-j. La implementación es la siguiente:

Ejemplo 11.26

```
function buscar : integer;
  var i,j:integer;
  begin
    i:= M;
    j:= M;
    {inicializar skip}
    repeat
      if a[i] = p[j] then begin
                          i:= i - 1;
                          j:= j - 1;
                          end
                          else begin
                            if M-j+1 > skip[index(a [i ])]
                                        then i:= i*M-j+1
                                        else i:= i+skip[index(a[i])];
                            j:= M;
                          end;
    until ( j < 1)  or  ( i > N);
    buscar := i + 1;
  end;
```

La instrucción i = i + M - j + 1 inicializa i en la próxima posición en el `string` de texto (como el patrón se desplaza desde izquierda a derecha a través de i), entonces j := M ubica el índice del patrón para prepararse para una coincidencia carácter de derecha a izquierda. La próxima instrucción desplaza el patrón a través del texto. Para el patrón `string`, la entrada a `skip` de G es cero, la entrada para N es 1, para I es 2, para T es 3, para S es 4 y las entradas para todas las otras letras debe ser 5. Esto, por ejemplo, indica que cuando se encuentra una S, el puntero i se debe incrementar en 4, esto es, el final del patrón se alínea 4 posiciones a la derecha de S.

Para aplicar el algoritmo de Boyer-Moore se debe tener en cuenta la siguiente propiedad nunca utiliza mas de M + N comparaciones y utiliza cerca de N / M pasos si el alfabeto no es pequeño y el patrón no es largo.

El algoritmo del caracter que no coincide, obviamente no ayuda mucho en el caso de *strings* binarios, porque solo se tienen dos posibilidades que pueden causar una no coincidencia (seguramente ambos deben estar en el patrón). Sin embargo, se puede hacer una agrupación, para tomar los bits como caracteres y utilizarlos como fue descripto anteriormente.

Conclusiones

Se ha profundizado el tratamiento de algoritmos, buscando analizar diferentes implementaciones para un mismo problema.

La conclusión más importante es que, dado el problema el especialista de software debe evaluar tanto las estructuras de datos como las estructuras de control usadas en el algoritmo para construir una solución eficiente.

Se ha puesto de manifiesto las ventajas y dificultades de las soluciones recursivas, dando un tratamiento general a las relaciones de recurrencia y su complejidad intrínseca.

Las nociones de complejidad algorítmica y eficiencia han sido tratadas casi como sinónimos, Estos conceptos se pueden profundizar para diferenciarlos, en la medida que se discutan otros modelos de máquina y otros modelos de especificaciones de algoritmos, lo que excede el alcance de este texto.

Referencias bibliográficas

Beetley, Haken, Saxe, "A general method for solving divide and conquer recurrences", Dept. of CS, Pittsburg, Carnegie Mellon University, 1978.

Fischer, "Efficiency of equivalence algorithms", Complexity of computer Computations, 1964.

Leeuwen, "Handbook of Theoretical Computer Sciencia", J. Van Leewen, Managing Editor, 1990.

Meyer, "Composite Structured Testing", Van Nostrand Reinhold, 1978.

Pipes, "Applied Mathematics for Engineers and Physicists", Mc Graw Hill, 1958.

Wirth, "Program development by stepwise refinement", Communications of the ACM, 1971.

12

Capítulo 12

Introducción al concepto de archivos

12

Introducción al concepto de archivos

Objetivo

En capítulos anteriores se han presentado diferentes estructuras de datos, cada una de ellas clasificadas de acuerdo a distintos criterios, que tienen en cuenta su organización interna. Sin embargo, todas estas estructuras tienen una particularidad en común: son definidas en un algoritmo, y ocupan memoria RAM, se utilizan en la ejecución y todos los valores contenidos en ellas se pierden, a lo sumo, cuando el algoritmo finaliza.

Para determinados problemas es necesario disponer de un tipo de estructura de datos que permita guardar su información en soporte no volátil (disco, cinta, zip, *diskette*), y de esa forma preservarla aunque el programa finalice. Estas estructuras de datos son conocidas, además, como estructuras de almacenamiento, y deben asociarse con un dispositivo de memoria auxiliar permanente donde archivan la información.

En este capítulo se presentan dichas estructuras, denominadas archivos, junto con las operaciones y algoritmos clásicos aplicables sobre los mismos.

12.1 Conceptos Generales

12.1.1 Memoria principal y memoria secundaria de una computadora

Las estructuras de datos que se definieron a lo largo del libro han utilizado la memoria interna de la computadora como lugar de almacenamiento. Esta memoria es denominada **principal** y normalmente es **memoria volátil de acceso aleatorio** (RAM: *Random Access Memory*).

Las principales características de la memoria principal o RAM se pueden resumir en:

- Existe un límite en la cantidad disponible. Si bien dicho límite es cada vez mayor con el avance de la tecnología, no existe una capacidad infinita. A mediados de los años 80 era normal tener computadoras con 640 KB de memoria RAM. A mediados de los años 90 dicha capacidad fue aumentada una 100 veces (64 MB) con costos muy inferiores. Hoy una computadora personal estándar posee 128 o 256 MB de RAM.
- Es memoria volátil: para conservar la información necesita electricidad, al apagar la computadora o ante un corte de energía su contenido se pierde.

Los dispositivos de **memoria secundaria** o **almacenamiento secundario** son medios de almacenamiento que están ubicados fuera de la memoria RAM, y que son capaces de retener información luego que el programa finaliza o de apagar la computadora. Algunos ejemplos de dispositivos de almacenamiento secundario son los discos rígidos, los discos flexibles (*diskettes*), discos ópticos, cintas.

La principal característica de la memoria principal es que, al ser eléctronica, su velocidad de acceso es muy elevada. Los dispositivos secundarios, en tanto, almacenan su información en un medio magnético (discos, cintas, diskettes) o en un medio óptico (CD, discos compactos), lo que hace más lenta su operación. Puede verse más en detalle el manejo interno de los medios de almacenamiento, por ejemplo en (Folk, 1992).

12.1.2 Tiempos de acceso

Como se dijo anteriormente, el almacenamiento secundario tiene la característica de ser más lento en el acceso que la memoria RAM. Si bien la velocidad de los dispositivos de memoria secundaria o auxiliar y de la memoria principal está ligada a las características físicas del medio, se puede decir que la RAM es del orden de 100.000 (cien mil) veces más rápida en su acceso que un disco rígido. El tiempo de acceso a memoria RAM está en el orden de nanosegundos (10^{-9} segundos), en tanto que el acceso a disco rígido está medido en milisegundos (10^{-3} segundos).

El lector podría llegar a la conclusión errónea de que, aunque la diferencia de tiempos sea muy grande, acceder a información contenida en un disco rígido no afecta en mucho la eficiencia de un algoritmo. Suponga por ejemplo 10000 lecturas de un disco rígido con un tiempo promedio de lectura de 15 milisegundos. Esto significa 150.000 milisegundos = 150 segundos = 2'30", lo que

no es un tiempo despreciable y posiblemente muy superior al de procesamiento o cálculo necesario sobre el dato.

12.1.3 Archivos

Los datos colocados en un dispositivo de almacenamiento secundario conforman lo que se denomina **archivo**. Se presentan dos definiciones:

Definición

Un archivo es una colección de registros semejantes, guardados en dispositivos de almacenamiento secundario de la computadora (Wiederhold, 1983).

Definición

Un archivo es una estructura de datos que guarda en un dispositivo de almacenamiento secundario de una computadora una colección de elementos del mismo tipo.

Estas definiciones muestran que un archivo es una estructura de datos homogénea. Los archivos se caracterizan por el crecimiento y las modificaciones que se efectúan sobre él. El crecimiento indica la incorporación de nuevos elementos y las modificaciones involucran alterar datos contenidos en el archivo, o quitarlos. Esto puede asemejarse a la forma de operación que tienen las estructuras de datos dinámicas definidas hasta el momento.

12.1.4 Administración de archivos

Cuando se trabaja con un archivo en disco rígido se hace referencia a un conjunto de datos almacenado en memoria secundaria. El disco rígido puede almacenar un gran número de archivos.

Sin embargo, desde un programa la noción de archivo no es exactamente la misma. Un programa no sabe con exactitud el lugar donde residirán los datos, o sea, el lugar dentro del disco rígido donde estará el archivo físico; el programa envía y recibe información, desde y hacia el archivo. Para poder efectuar esto, el programa de aplicación se apoya en el sistema operativo de la computadora, que es el encargado de los detalles de almacenamiento, tales como lugar dentro del dispositivo físico, cantidad de espacio que va a utilizar, tipos de acceso permitido (si se puede leer, leer y escribir, si el acceso esta vedado, etc.), usuarios que tienen acceso a los datos del archivo, etc. El programa referencia indirectamente al archivo físico utilizando para ello un nombre lógico, que se denomina **archivo lógico**.

Antes de que el programa pueda operar sobre el archivo, el sistema operativo debe recibir instrucciones para hacer un enlace entre el nombre lógico que utilizará el algoritmo y el archivo físico. Cada lenguaje de programación define una instrucción para tal fin.

12.1.5 Operaciones básicas sobre archivos

Se describen en esta sección una serie de operaciones elementales que son necesarias para trabajar con archivos. Estas operaciones básicas son cinco:

- Vincular o relacionar el archivo físico (que está en memoria secundaria) con el nombre lógico que se utilizará dentro del algoritmo.
- Abrir un archivo para su procesamiento.
- Cerrar un archivo, luego de operar con el mismo.
- Leer los elementos contenidos en un archivo.
- Escribir nuevos elementos en un archivo o modificar algunos existentes.

12.1.5.1 Relación con el Sistema Operativo

Como se dijo anteriormente, dentro de un programa se trabaja con un archivo lógico, el cual está relacionado con un archivo físico que reside en memoria secundaria. Se debe disponer, pues, de una operación que permita establecer la relación que debe existir entre el archivo físico con el archivo lógico. Cada lenguaje de programación presenta una opción para esto, en líneas generales el formato será:

```
Asignar_correspondencia( nombre_fisico, nombre_logico)
```

El nombre físico define exactamente el nombre con el que el sistema operativo encontrará al archivo dentro del almacenamiento secundario. En tanto, el nombre lógico se corresponde con una variable definida en el algoritmo. Dicha variable debe ser de tipo archivo, dependiendo su definición del lenguaje utilizado.

12.1.5.2 Apertura de archivos

Si bien para trabajar con un archivo en un algoritmo el primer paso es establecer la relación entre el nombre lógico y el físico, esta operación no es suficiente para poder operar con dicho archivo. Se debe realizar la apertura del mismo indicando el tipo de operaciones que se van a realizar. Estas operaciones pueden ser: de **solo lectura** o **lectura y escritura**. Los lenguajes de programación poseen instrucciones específicas para tal fin, en todos los casos, el parámetro necesario es el nombre lógico del archivo.

La apertura de un archivo establece una posición corriente en el mismo, la cual indica el dato que va a ser leído o el lugar sobre el que va a ser escrito. Si bien cada lenguaje maneja esta posición de manera diferente, todos la utilizan para poder operar con el archivo.

12.1.5.3 Cierre de archivos

Así como un archivo debe ser abierto para poder trabajar con el mismo, una vez finalizado el trabajo debe ser cerrado. Esta operación de cierre determina la desaparición de la posición corriente del archivo y agrega al final del mismo una marca conocida como fin de archivo (**eof**, *end of file*).

En todos los casos, es aconsejable efectuar el cierre del archivo con la instrucción provista por el lenguaje. Sin embargo, cuando se llega al final del módulo que realizó la apertura del archivo, en la mayoría de los lenguajes de programación, el archivo se cierra automáticamente.

12.1.5.4 Lectura y escritura sobre archivos

Hasta aquí los archivos fueron definidos, relacionados (nombre físico, nombre lógico), abiertos para su operación y posteriormente cerrados. Falta ,ahora, poder agregar información o leer los datos que contiene.

Cada lenguaje de programación presenta dos instrucciones con, básicamente, el siguiente formato:

- Leer (archivo, dato)
- Escribir (archivo, dato)

donde **archivo** corresponde al nombre lógico utilizado para el mismo, y **dato** corresponde a una variable que debe ser del mismo tipo para el cual el archivo fue definido. Esto es, si se declara un archivo que contendrá números enteros, la variable debe ser de este tipo; si el archivo almacenara registros de un tipo previamente definido, la variable debe ser de dicho tipo. En cualquier caso, la operación se efectúa sobre la posición corriente en el archivo. Esto es, se lee o escribe desde, o hacia, el lugar indicado.

12.1.6 Técnicas de organización y acceso a un archivo

Según el modo en que se organizan los registros dentro de un archivo, se consideran dos tipos de acceso a los registros de un archivo:

- Acceso secuencial.
- Acceso directo.

El acceso secuencial permite acceder a los registros o elementos uno tras otro y en el orden físico en que están guardados. En cambio, el acceso directo permite obtener un registro determinado sin necesidad de haber accedido a sus predecesores.

De acuerdo a su organización un archivo puede clasificarse como:

- Secuencial.
- Directo.
- Secuencial indizado.

esta organización define la manera en que los registros se distribuyen sobre el almacenamiento secundario.

Un archivo secuencial consiste de un conjunto de registros almacenados consecutivamente de manera que para acceder al registro n-ésimo del mismo se debe, previamente, acceder a los n-1 registros anteriores. Los registros se graban en forma consecutiva, a medida que se ingresan, y se recuperan en el mismo orden.

Un archivo directo consiste de un conjunto de registros donde el ordenamiento físico no necesariamente corresponde con el ordenamiento lógico. Esto es, los registros se recuperan accediendo por su posición dentro del archivo. Por lo tanto, es posible acceder al n-ésimo lugar sin haber accedido a los n-1 registros anteriores. Esta organización presenta la ventaja que se puede obtener cualquier elemento del archivo en cualquier orden, siendo muy eficientes en cuanto a tiempo de acceso necesario para recuperar la información. Por otro lado, presentan el inconveniente de tener que determinar la posición donde se encuentra cada elemento. Por ejemplo, alguna función que dada la identificación del registro retorne su posición en el disco físico (Smith, 1998). Además, se debe contar, dentro del archivo, con espacio disponible para poder ingresar nuevos elementos.

Un archivo secuencial indizado utiliza estructuras de datos auxiliares para permitir un acceso pseudo directo a los registros de un archivo. Un ejemplo de esta organización puede ser una guía telefónica. En ella se puede acceder por letra, y dentro de cada página existe una indicación de apellido de comienzo y apellido de fin dentro de la hoja; de esta forma se puede acotar el espacio de búsqueda de un determinado teléfono dentro de la guía, haciendo referencia mucho más rápidamente a la hoja donde se encuentra el dato. Los archivos organizados con esta técnica tienen la ventaja de tener un acceso mucho más rápido que los secuenciales, pero necesitan más espacio para mantener la/s estructura del/los índice/s. Las estructuras de índice se suelen denominar directorios del archivo.

En este libro se presentarán algoritmos que utilizan archivos organizados secuencialmente, y, en la última sección, ejemplos donde se utilicen archivos secuenciales indizados. El lector puede encontrar en otra bibliografía clasificaciones de organización de archivos que respondan a las definiciones anteriores pero que reciban otro nombre para referenciarlas.

12.1.7 Manejo de buffers

Antes de comenzar el desarrollo se define el concepto de *buffer*.

Definición

Se denomina *buffer* a una memoria intermedia entre un archivo y un programa, donde los datos residen provisoriamente hasta ser almacenados definitivamente en memoria secundaria o donde los datos residen una vez recuperados de dicha memoria secundaria. Los *buffers* ocupan una zona de la memoria RAM de la computadora. Figura 12.1

Manejar *buffers* implica trabajar con grupos de datos en memoria RAM para que el número de accesos al almacenamiento secundario se reduzca. Básicamente, las operaciones de lectura

(*read*) y escritura (*write*) no se realizan directamente sobre la memoria secundaria. Si esto fuera así, se necesitaría, ante cada una de estas operaciones una determinada cantidad de milisegundos para realizarla. El procesamiento de archivos sería así, un trabajo muy tedioso por el tiempo involucrado. Por lo tanto, las instrucciones `read` y `write`, que se definen en los algoritmos, interactúan con un *buffer*, el cual al encontrarse en memoria RAM agilizan el proceso.

El sistema operativo de la computadora es el encargado de manipular los *buffers*, de la siguiente manera. Cuando un programa realiza una operación de lectura, y en el *buffer* no hay información para satisfacerla, se lee de memoria secundaria los datos para completar nuevamente dicho *buffer* y así poder satisfacer el requerimiento. Algo similiar ocurre con la escritura, cuando se intenta escribir en un *buffer* y el mismo no tiene más capacidad, la información contenida es bajada o guardada en memoria secundaria, una vez vacío el *buffer* puede tomar los datos definidos para `write`.

Normalmente, el sistema operativo asigna a cada archivo un *buffer* de entrada y otro de salida. Figura 12.1.

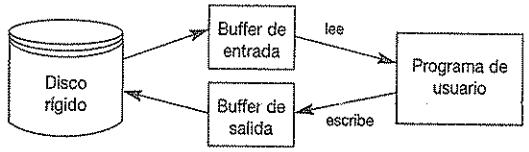


Figura 12.1

12.2 Operaciones básicas sobre archivos en Pascal

Pascal, como cualquier lenguaje de programación, tiene instrucciones para cada una de las operaciones previamente definidas. En esta sección se presenta cada una de ellas, comenzando con las necesarias para declarar los archivos en un algoritmo.

12.2.1 Declaración de archivos

Los archivos, en el lenguaje Pascal, pueden definirse utilizando dos formas:

```
Var archivo: file of tipo_de_dato;
```

donde `tipo_de_dato` debe ser algún tipo de dato definido previamente y sobre el cual se puedan realizar operaciones de entrada y salida.

```
Type  archivo: file of tipo_de_dato;
Var     arch: archivo;
```

En este caso, se define un nuevo tipo de datos denominado archivo, y luego se definen variables de ese nuevo tipo de datos. Note que las dos definiciones son similares a los casos planteados para arreglos; si bien ambas son correctas, la segunda definición debe utilizarse para poder enviar o recibir como parámetro al archivo.

Ejemplo 12.1

Este programa presenta la definición de dos archivos.

```
type emp = record;
        nombre: string[20];
        dirección: string[20];
        edad: integer;
        salario: real;
      end;
      numeros = file of integer;
      empleados = file of  emp;
var archivo_de_numero: numero;
    archivo_de_empleados: empleados;
```

archivo_de_numero determina un archivo que contendrá números enteros, en tanto que archivo_de_empleados contendrá los datos personales de los empleados de alguna empresa.

12.2.2 Relación con el sistema operativo

El formato de la instrucción Pascal que permite establecer la correspondencia entre el nombre físico y el lógico de un archivo es el siguiente:

```
Assign( nombre_logico, nombre_fisico);
```

donde `nombre_logico` debe ser una variable de tipo `file` definida como variable.

Ejemplo 12.2

El siguiente segmento de código, muestra las instrucciones necesaria para definir un archivo que contenga números enteros y que en dispositivo de almacenamiento secundario se encontrará bajo el nombre de "numeros.dat".

```
Program Definir_archivo;
  .......
  type archivo_numerico: file of  integer;
        .......
  var an: archivo_numÈrico;
        .......
  begin
```



```
      assign( an, 'numeros.dat');
      .......
  end;
```

Generalmente, dentro del algoritmo la correspondencia entre el nombre físico y lógico se efectúa al comienzo del mismo y dicha relación es válida para toda la duración del algoritmo.

12.2.3 Apertura y creación de archivos

Como se dijo anteriormente, un archivo puede abrirse como de lectura y escritura, o de lectura solamente. La primer definición opera sobre un archivo existente en memoria secundaria, en tanto que la segunda tiene como objetivo crear dicho archivo. Pascal presenta dos instrucciones, una para cada fin, estas son:

```
Reset( nombre_lógico); { abre un archivo existente como de lectura y escritura }

Rewrite( nombre_lógico); { abre el archivo por primera vez, sólo para escritura }
```

en ambos casos `nombre_lógico` representa la variable de tipo archivo sobre la que se realizó la asignación correspondiente. Nótese que si se realiza la apertura con la instrucción `Rewrite` y el archivo contiene información, la misma se pierde totalmente.

12.2.4 Cierre de archivos

Pascal define la instrucción:

```
Close( nombre_lógico );
```

para efectuar el cierre explícito de un archivo y colocar la marca que indica su finalización.

12.2.5 Lectura y escritura en archivos

Las operaciones de entrada y salida desde y hacia un archivo son similares a las operaciones de lectura de información de teclado o a la escritura en la pantalla de la computadora. Las instrucciones son:

```
Read( nombre_logico, variable );
Write( nombre_logico, variable);
```

donde `nombre_logico` se corresponde con el nombre lógico del archivo y `variable` es una variable cuyo tipo de dato debe corresponderse con la definición del archivo. Cada una de estas instrucciones opera sobre la posición actual del archivo, y luego avanza a la posición siguiente.

Ejemplo 12.3

El siguiente algoritmo, presenta la creación de un archivo compuesto por números enteros, dichos números enteros se obtienen desde el teclado hasta obtener el valor 0 el cual no debe ser parte del archivo.

Precondiciones:

- Hay espacio suficiente en la memoria secundaria para generar el archivo.
- El nombre del archivo se obtiene desde teclado.

Poscondiciones:

- El archivo queda efectivamente guardado en el disco rígido.

```
Program Generar_Archivo;
  type archivo = file of integer; {definción del tipo de dato para
                                   el archivo}
  var arc_logico: archivo; {variable que define el nombre lógico
                            del archivo}
      nro: integer; {nro será utilizada para obtener la información
                     de teclado}
      arc_fisico: string[12]; {utilizada para obtener el nombre
                               físico del archivo desde teclado}

  begin
    write( 'Ingrese el nombre del archivo:' );
    read( arc_fisico );      {se obtiene el nombre del archivo}
    assign( arc_logico, arc_fisico );
    rewrite( arc_logico );       {se crea el archivo}
    read( nro );             {se obtiene de teclado el primer valor}
    while nro <> 0 do
      begin
        write( arc_logico, nro );{se escribe en el archivo cada número}
        read( nro );
      end;
    close( arc_logico );  {se cierra el archivo abierto
                           oporturnamente con la instrucción rewrite}
  end.
```

12.2.6 Operaciones adicionales

Si bien con las operaciones tratadas hasta el momento se puede definir y crear un archivo; es imposible recorrer, agregar o modificar la información contenida. Entonces, son necesarias una serie de operaciones adicionales que permiten manipular los archivos existentes para efectuar las operaciones anteriormente descritas.

Para recorrer un archivo desde el primer registro y hasta el final, es necesario contar con una operación que detecte dicho fin de archivo. La sintaxis en Pascal de la función que detecta este fin de archivo es:

```
Eof( nombre_logico );
```

donde `nombre_logico` se corresponde con el nombre lógico del archivo. Esta función retorna verdadero si la posición corriente dentro del archivo referencia a la marca de fin, falso en caso contrario.

Suponga que ha creado un archivo con los datos personales de empleados de una empresa. Si se necesita saber el número de registros cargados (o sea la cantidad de empleados), se debe contar con una instrucción que retorne el número de elementos del archivo, para hacer más eficiente el procesamiento. En Pascal dicha función es la siguiente:

```
Filesize( nombre_logico) ;
```

donde `nombre_logico` se corresponde con el nombre lógico del archivo. Esta función retorna un número entero, que corresponde a la cantidad de registros del archivo.

Otros procesos requieren operar con la posición corriente en el archivo, para lo cual es necesario poder determinarla.

```
Filepos( nombre_logico );
```

retorna un número entero correspondiente a la posición actual del apuntador en el archivo.

Con la instrucción anterior se puede determinar la posición corriente del archivo. Para modificar el contenido de esta dirección es necesaria otra operación. Pascal provee la instrucción `seek` la cual permite llegar a un elemento particular del archivo. La sintaxis completa de la misma es la siguiente:

```
Seek( nombre_logico, posición );
```

indicando `nombre_logico` el nombre lógico del archivo, como en las operaciones anteriores, en tanto que `posición` es un valor entero comprendido entre 0 (primer elemento del archivo) y n-1 (suponiendo que n es un número máximo de elementos del archivo).

Ejemplo 12.4

Siguiendo con las declaraciones efectuadas en el ejemplo anterior, realizar un procedimiento que reciba al archivo anteriormente generado y presente el contenido del mismo en pantalla.

Precondiciones:

- Las declaraciones de tipos necesarias están en el programa principal.
- El archivo puede contener 0, o más elementos.

Poscondiciones:

- Se recorrió todo el archivo imprimiendo cada elemento.

```
Procedure Recorrido(var arc_logico: archivo );
  var  nro: integer;          {para leer cada elemento del archivo}
  begin
    reset( arc_logico );{el archivo está creado, para operar con el
                          mismo debe abrirse como de lectura escritura}
    while not eof( arc_logico) do
      begin
        read( arc_logico, nro ); {se obtiene cada elemento desde el
                                  archivo}
        write( nro ); {se presenta cada valor en pantalla}
      end;
    close( arc_logico );
  end;
```

Este procedimiento presenta dos aspectos interesantes para considerar. El primero de ellos, los parámetros correspondientes a nombre lógico del archivo deben, en Pascal, necesariamente enviarse por referencia. El segundo aspecto, también relacionado con Pascal, es que antes de efectuar una operación de lectura de un archivo se debe evaluar necesariamente el fin de archivo.

12.3 Algoritmos clásicos sobre archivos

En esta sección se discuten algunos casos considerados usuales en el manejo de archivos:

- Modificar el contenido actual.
- Agregar nuevos elementos.
- Actualización de un archivo maestro con uno o varios archivos detalle.
- Corte de control.
- *Merge* o unión de varios archivos.

12.3.1 Caso 1: Modificación de datos de un archivo

Este caso involucra un archivo de datos previamente generado, y consiste en modificar sus datos. Para ello el archivo debe ser recorrido desde el primer elemento hasta el último, siguiendo un procesamiento secuencial del archivo.

Ejemplo 12.5

En el siguiente ejemplo se supone la existencia del archivo, el cual se corresponde con la estructura definida a continuación, la cual representa los datos personales de empleados de una empresa, junto con el sueldo que reciben. El procedimiento de actualización aumenta un 10% el salario de cada empleado contenido en el archivo.

Precondiciones:

- El archivo ya existe.
- La estructura del archivo esta definida antes del procedimiento y en el programa principal.
- La asignación nombre físico, nombre lógico está realizada en el programa principal.
- El archivo no está abierto.

Poscondiciones:

- El archivo queda cerrado y actualizado.

```
{declaración de los tipos de datos necesarios para el problema
 Esta declaración se hace efectiva en el programa principal que
 tiene al proceso Actualizar como uno de sus módulos}

Type registro = record
        Nombre: string[20];
        Direccion: string[20];
        Salario: real;
      End;
      Empleados = file of registro;
...
...

Procedure actualizar (Var Emp: empleados);  {se recibe el archivo
                                             como parámetro por referencia}
  var E: registro;
  begin
    Reset( Emp );      {el archivo contiene datos, se abre de E/S }
    while not eof( Emp ) do
      begin
        Read( Emp, E);        {se obtiene el elemento del archivo}
        E.salario := E.salario * 1.1; {se incrementa el salario en un 10%}
        Seek( Emp, filepos(Emp) -1 );
        {luego de la lectura la posición corriente del archivo
         avanza unaposición, para hacer la escritura se debe
         retroceder en uno dicha posición}
        Write( Emp, E );      {se escribe el registro modificado }
      end;
    close( Emp );
  end;
```

Nótese que, como se dijo en el ejemplo anterior, la lectura del archivo se realiza luego de evaluar si lo que sigue no es la marca la finalización del mismo. Además, en este caso, se observa que cada operación de lectura o escritura sobre un archivo, en Pascal, modifica la posición corriente, por lo tanto para poder guardar las actualizaciones realizadas sobre el archivo, necesariamente se debe retroceder dicha posición corriente.

12.3.2 Caso 2: Agregado de datos a un archivo

Nuevamente, este caso presenta el procesamiento sobre un solo archivo. Un problema muy común es agregar nuevos registros, siguiendo un procesamiento secuencial, que se agregan al final del mismo (*Append*).

Ejemplo 12.6

Se presenta a continuación el problema de agregar nuevos datos a un archivo existente. Este problema utilizará las definiciones del ejercicio anterior.

Precondiciones:

- El archivo ya existe.
- La estructura del archivo está definida antes del procedimiento y en el programa principal.
- La asignación nombre físico, nombre lógico está realizada en el programa principal.
- El archivo no está abierto.
- Hay suficiente espacio en la memoria secundaria para contener más datos que se agregan al archivo.

Poscondiciones:

- El archivo queda cerrado y actualizado.

```
Procedure agregar (Var Emp: empleados); {se recibe el archivo como
                                          parámetro por referencia}
  var E: registro;
  begin
    reset( Emp );     {el archivo contiene datos, se abre de E/S}
    seek( Emp, filesize(Emp));  {avanza la posición corriente
                                 hasta el final del archivo,
                                 donde está la marca de EOF}
    leer( E );  {procedimiento que obtiene un registro
                 (paraincorporar al archivo)}
    while E.nombre <> ' ' do
      begin
        write( Emp, E );      { se escribe el nuevo registro }
        leer( E );            { se obtiene el siguiente elemento }
      end;
    close( Emp );
  end;
```

En este ejercicio se plantea la necesidad de agregar nuevos empleados a un archivo ya existente. Si se intentara escribir luego de realizar la apertura del archivo, se perderían los primeros empleados, ya que, luego de la apertura, la posición actual dentro del archivo referencia al primer empleado. La instrucción `seek` efectúa un desplazamiento de la posición de trabajo, desde el comienzo hacia el último elemento del archivo (debido a que se saltean tanto registros como indica el tamaño del archivo), y este último elemento es la indicación de fin de archivo, la cual es quitada de esa posición, para luego agregarse en el nuevo fin del archivo (con la instrucción `close`).

12.3.3 Caso 3: Actualización de un archivo maestro con uno o varios archivos detalle

Este tipo de problemas ya involucra el procesamiento simultáneo de varios archivos. Se denomina archivo maestro a aquel archivo que resume un determinado conjunto de datos, en tanto que se denomina archivo/s detalle/s a aquel/los que agrupan información que se utilizará para modificar el contenido del archivo maestro. Normalmente se dispone de un archivo maestro y n archivos detalles relacionados con un problema particular.

Ejemplo 12.7

El problema a resolver consiste en actualizar un archivo maestro que contiene información sobre los empleados de una empresa: nombre, dirección, cantidad de horas trabajadas. Este archivo se encuentra generado y los empleados aparecen, en el mismo, ordenados alfabéticamente. Diariamente se genera un archivo de detalles, el cual posee el nombre del empleado y la cantidad de horas trabajadas ese día. Este archivo también se encuentra ordenado alfabéticamente. El proceso que se presenta actualiza las horas trabajadas por el empleado con la información contenida en el detalle. Dentro de las precondiciones del problema se presentan otras características a tener en cuenta.

Precondiciones:

- Ambos archivos están generados y se encuentran ordenados alfabéticamente.
- En el archivo de detalle solo aparecen empleados que existen en el archivo maestro.
- Cada empleado del archivo maestro a lo sumo puede aparecer una vez en el archivo de detalle.

Poscondiciones:

- El archivo maestro queda actualizado con la información contenida en el archivo detalle.

```
program actualizar;
  type emp = record
          nombre: string[30];        { nombre del empleado }
          direccion: string[30];     { dirección del empleado }
          cht: integer;              { cantidad de horas trabajadas }
        end;
        e_diario = record
          nombre: string[30];        { nombre del empleado }
          cht: integer;   {cantidad de horas trabajadas en el día}
        end;
        detalle = file of e_diario; {archivo que contiene la
                                     información diaria }
        maestro = file of emp;  {archivo que contiene la información
                                 completa}
  var
     regm: emp;
     regd: e_diario;
```

```
    mae1: maestro;
    det1: detalle;
  begin
    assign (mae1, 'maestro');
    assign (det1, 'detalle');
    {proceso principal}
    reset (mae1);
    reset (det1);
    while (not eof(det1)) do
      begin
        read(mae1, regm);
        read(det1,regd);
        while (regm.nombre <> regd.nombre) do
          read (mae1,regm);
        regm.cht := regm.cht + regd.cht;
        seek (mae1, filepos(mae1)-1);
        write(mae1,regm);
      end;
  end.
```

En este caso, el procesamiento finaliza luego de recorrer todo el archivo detalle, dado que el archivo maestro se actualiza con dicha información. Es por este motivo que en la iteración principal se controla el fin de archivo detalle, ignorando al archivo maestro. Además, las lecturas sobre el archivo maestro se efectúan sin controlar su `eof`, debido a una de las precondiciones establecidas donde se indica que en el archivo detalle solo aparecen empleados que existen en el archivo maestro.

Ejemplo 12.8

Este problema generaliza al caso anterior. Ahora el archivo maestro tiene la siguiente estructura: código de producto, descripción del producto, precio unitario y cantidad existente del producto. En el archivo detalle figuran las ventas realizadas en un día, la estructura es la siguiente: código del producto, cantidad vendida. Ambos archivos se encuentran ordenados por código del producto.

Precondiciones:

- Ambos archivos están generados y se encuentran ordenados por código del producto.
- En el archivo de detalle solo aparecen productos que existen en el archivo maestro.
- Cada producto del archivo maestro puede ser, a lo largo del día, vendido más de una vez, por lo tanto, en el archivo detalle pueden existir varios registros correspondientes al mismo producto.

Poscondiciones:

- El archivo maestro queda actualizado con la información contenida en el archivo detalle.


```
program actualizar;
  const valoralto='9999';
  type str4 = string[4];
       prod = record
         cod: str4;                  {código del producto}
         descripcion: string[30];    {descripción del producto}
         pu: real;                   {precio unitario del producto}
         cant: integer;              {cantidad existente del producto}
       end;
       v_prod = record
         cod: str4;                  {código del producto}
         cant_vendida: integer;      {cantidad vendida del producto}
       end;
       detalle = file of v_prod;     {archivo que contiene la
                                      información diaria}
       maestro = file of prod;       {archivo que contiene la información
                                      completa}
  var
    regm: prod;
    regd: v_prod;
    mae1: maestro;
    det1: detalle;
    total: integer;
    aux: str4;

  procedure leer (var archivo:detalle; var dato:v_prod);
    begin
      if (not eof(archivo))
        then read (archivo,dato)
        else dato.cod := valoralto;
    end;

  begin
    assign (mae1, 'maestro');
    assign (det1, 'detalle');
    {proceso principal}
    reset (mae1);
    reset (det1);
    read(mae1,regm);
    leer(det1,regd);
    {se procesan todos los registros del archivo det1}
    while (regd.cod <> valoralto) do
      begin
        aux := regd.cod;
        total := 0;
        {se procesan todos los registros del mismo producto}
        while (aux = regd.cod ) do
          begin
            total := total + regd.cant_vendida;
            leer(det1,regd);
          end;
```

```
        {se localiza el registro del archivo maestro correspondiente}
         while (regm.cod <> aux) do
           read (mae1,regm);
         regm.cant := regm.cant - total;
         {reubica el puntero en el registro correspondiente}
         seek (mae1, filepos(mae1)-1);
         write(mae1,regm);
         if (not eof (mae1))
           then read(mae1,regm);
       end;

    { ver resultados }
    reset (mae1);
    while (not eof(mae1)) do
      begin;
        read (mae1, regm);
        writeln (regm.cod, regm.cant);
      end;
  end.
```

En líneas generales, el presente proceso es similar al planteado anteriormente. El procedimiento finaliza luego de procesar todo el archivo detalle. La diferencia radica en que, ahora, una vez localizado el registro del maestro que tiene modificaciones, puede sufrir varios cambios antes de guardarse nuevamente.

Ejemplo 12.9

Por último, generalizando aún más el problema anterior, se actualiza ahora el archivo maestro con varios archivos detalles. Con el fin de facilitar la comprensión del problema, se acota a tres la cantidad de archivos de detalle, aunque el mismo proceso puede generalizarse para n archivos con muy pocos cambios. Las estructuras de datos necesarias para implementar la solución son las descriptas en el ejercicio anterior. Asimismo, las precondiciones y poscondiciones son similares. *A priori*, la solución del problema podía consistir en invocar al proceso del ejemplo anterior tres veces, enviando como parámetro el archivo maestro y uno de los archivos detalle por vez. Es una solución correcta pero ineficiente. Nótese que el archivo maestro es procesado o recorrido tres veces por lo cual el tiempo de ejecución crece considerablemente. La solución presentada a continuación procesa simultáneamente los tres archivos detalles.

```
program Cap11_31;
  const valoralto='9999';
  type str4 = string[4];
       prod = record
         cod: str4;                  { código del producto }
         descripcion: string[30];    { descripción del producto }
         pu: real;                   { precio unitario del producto }
         cant: integer;              { cantidad existente del producto }
       end;
```



```
       v_prod = record
         cod : str4;                 { código del producto }
         cant_vendida: integer;      { cantidad vendida del producto }
       end;
       detalle = file of v_prod;     { archivo que contiene la
                                       información diaria}
       maestro = file of prod;       { archivo que contiene la
                                       información completa }

var
  regm: prod;
  min, regd1, regd2, regd3: v_prod;
  mae1: maestro;
  det1,det2,det3: detalle;
  aux: str4;

procedure leer (var archivo:detalle; var dato:v_prod);
  begin
    if (not eof(archivo))
      then read (archivo,dato)
      else dato.cod := valoralto;
  end;

procedure minimo (var r1,r2,r3:v_prod; var min:v_prod);
  begin
    if (r1.cod<=r2.cod) and (r1.cod<=r3.cod)
      then begin
        min := r1;
        leer(det1,r1)
      end
      else if (r2.cod<=r3.cod)
             then begin
               min := r2;
               leer(det2,r2)
             end
             else begin
               min := r3;
               leer(det3,r3)
             end;
  end;
begin
  assign (mae1, 'maestro');
  assign (det1, 'detalle1');
  assign (det2, 'detalle2');
  assign (det3, 'detalle3');
  {proceso principal}
  reset (mae1);
  reset (det1);
  reset (det2);
  reset (det3);
```

```
    read(mae1,regm);
    leer(det1, regd1);
    leer(det2, regd2);
    leer(det3, regd3);
    minimo(regd1,regd2,regd3,min);

    while (min.cod <> valoralto) do
      begin
        while (regm.cod <> min.cod) do
          read(mae1,regm);
        aux := min.cod;
        while (aux = min.cod ) do
          begin
            regm.cant := regm.cant - min.cant_vendida;
            minimo(regd1,regd2,regd3,min);
          end;
        seek (mae1, filepos(mae1)-1);
        write(mae1,regm);
        if (not eof (mae1))
          then read(mae1,regm);
      end;
    { ver resultados }
    reset (mae1);
    while (not eof(mae1)) do
      begin;
        read (mae1, regm);
        writeln (regm.cod, regm.cant);
      end;
  end.
```

El objetivo del problema es la actualización del archivo maestro, sin importar el número de archivos detalles. En particular, la solución presentada secuencializa la información proveniente de los archivos detalles, esto es, sin importar de donde provienen los registros de actualización se tratan en orden de menor a mayor. Para esto se cuenta con el procedimiento mínimo, que recibe un registro de cada detalle y retorna aquel que tenga código mínimo. Tenga en cuenta el tratamiento del fin de proceso, el mismo finaliza luego de procesar los tres archivos detalles, se utiliza para ello una marca, consistente en asignar un valor inexistente al código de producto, una vez que se detecta que los archivos detalles finalizaron.

12.3.4 Caso 4: El problema del "corte de control"

El siguiente tipo de problemas clásicos consiste en la generación de reportes (informes impresos) a partir de información contenida en un archivo, siguiendo para ello algunas pautas definidas.

Dado que "corte de control" es, nuevamente, un caso práctico; para definirlo se debe recurrir a la descripción de sus principales características de uso.

Ejemplo 12.10

El Instituto de Estadísticas y Censos dispone de información que resume datos sobre la población de Argentina. La información está categorizada por ciudad, partido o departamento y provincia, conociendo en cada caso la cantidad de varones y mujeres y la cantidad de personas sin trabajo. Esta información está resumida en un archivo el cual se encuentra ordenado por provincia, dentro de cada provincia por partido o departamento.
A pedido de la Dirección del Instituto, se debe obtener un listado resumen, el cual debe tener el siguiente formato:

Provincia: La Pampa
Partido o Departamento: Mara Có

Ciudad	Número Varones	Número Mujeres	Desocupados
General Pico	... ...	...	
Dorila ...	... ...		
Total Partido	... ...	...	

Partido o Departamento: Capital

Ciudad	Número Varones	Número Mujeres	Desocupados
Santa Rosa	...	...	...
Anguil	...	...	...
Total Partido	...	...	...
Total Provincia	...	...	...

Provincia: Cordoba
Partido o Departamento: ...
Ciudad Número Varones Número Mujeres Desocupados

Precondición:

- El archivo está ordenado por provincia y partido o departamento

Poscondición:

- Se obtiene el listado anterior.

```
program Corte_de_Control;
  const valoralto='zzzz';
  type str10 = string[10];
       prov = record
          provincia: str10;          { nombre provincia }
          partido: str10;            { nombre partido o Depto }
          ciudad: str10;             { nombre de ciudad }
```

```
          cant_varones : integer;        { cantidad de varones }
          cant_mujeres : integer;        { cantidad de mujeres }
          cant_desocupados : integer;    { cantidad de desocupados }
        end;
        instituto = file of prov;        { archivo que contiene la
                                           información completa }

  var regm: prov;
      inst: instituto;
      t_varones, t_mujeres, t_desocupados: integer;
      t_prov_var, t_prov_muj, t_prov_des: integer;
      ant_prov: str10;
      ant_partido: str10;

  procedure leer (var archivo:instituto; var dato:prov);
    begin
      if (not eof( archivo ))
        then read (archivo,dato)
        else dato.provincia := valoralto;
    end;
begin
    assign (inst, 'censo' );

    {creacion de archivos}
    reset (inst);
    leer (inst, regm);
    writeln ('Provincia: ', regm.provincia);
    writeln ('Partido: ', regm.partido);
    writeln ('Ciudad','Varones','Mujeres','Desocupados');
     { se inicilizan los contadores para el total del partido para
      varones, mujeres y desocupados }
    t_varones := 0;
    t_mujeres := 0;
    t_desocupados := 0;
     {se iniciliazan los contadores para el total de cada provincia}
    t_prov_var := 0;
    t_prov_muj := 0;
    t_prov_des := 0;

    while ( regm.provincia <> valoralto)do
      begin
        ant_prov := regm.provincia;
        ant_partido := regm.partido;
        while (ant_prov=regm.provincia) and (ant_partido=regm.partido) do
         begin
           write (regm.ciudad,regm.cant_varones,regm.cant_mujeres,
                  regm.cant_desocupados);
```



```
           t_varones := t_varones + regm.cant_varones;
           t_mujeres := t_mujeres + regm.cant_mujeres;
           t_desocupados := t_desocupados + regm.cant_desocupados;

           t_prov_var := t_prov_var + regm.cant_varones;
           t_prov_muj := t_prov_muj + regm.cant_mujeres;
           t_prov_des := t_prov_des + regm.cant_desocupados;

           leer (inst, regm);
         end;
       write ('Total Partido: ', t_varones,t_mujeres,t_desocupados);
       writeln;
       t_varones := 0;
       t_mujeres := 0;
       t_desocupados := 0;
       ant_partido := regm.partido;
       if (ant_prov <> regm.provincia)
         then begin
           write ('Total Provincia: ', t_prov_var,t_prov_muj,
                  t_prov_des);
           writeln;

           t_prov_var := 0;
           t_prov_muj := 0;
           t_prov_des := 0;
           writeln ('Provincia: ', regm.provincia);
         end;
       writeln ('Partido: ', regm.partido);
     end;
end.
```

12.3.5 Caso 5: El problema de la unión de archivos *(merge)*

El último caso práctico que se presenta en esta sección involucra varios archivos conteniendo información similar, el cual debe resumirse en un único archivo. Las suposiciones para plantear la solución a este problema parten de tener archivos secuenciales ordenados físicamente todos por un criterio similar. Esta operación de agrupamiento de información puede verse como un proceso de decisión con respecto a cuál de los datos provenientes de los archivos representan la misma información (que debe agruparse) o, en caso que no la representen, cual tiene la información con valor mínimo de acuerdo al criterio de orden para los archivos.

A continuación se presentan dos casos prácticos, en cada uno de ellos se plantean características particulares del problema.

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

Ejemplo 12.11

Programación de computadoras inscribe a los alumnos que cursarán la materia en tres computadoras separadas. Cada una de ellas genera un archivo con los datos personales de los estudiantes, que luego son ordenados físicamente por otro proceso. El problema que tienen los Jefes de Trabajos Prácticos es juntar los tres archivos para conformar el archivo maestro de la asignatura. Este programa presenta una solución al problema.

Precondiciones:

- El proceso recibe tres archivos con igual estructura.
- Los archivos están ordenados por nombre del alumno.
- Un alumno solo aparece una vez en el archivo.

Poscondición:

- Se genera el archivo maestro de la asignatura ordenado por nombre del alumno.

```
program union_de_archivos;

  const valoralto = 'zzzz';
  type str30 = string[30];
       str10 = string[10];
       alumno = record
         nombre: str30;          {nombre y apellido del alumno}
         dni: str10;             {dni del alumno}
         direccion: str30;       {dirección del alumno}
         carrera: str10;         {carrera que cursa el alumno}
       end;
       detalle = file of alumno; {archivo que contiene información
                                  de alumnos }

  var min, regd1, regd2, regd3 : alumno;
      det1, det2, det3, maestro : detalle;

  procedure leer (var archivo:detalle; var dato:alumno);
    begin
      if (not eof( archivo ))
        then read (archivo, dato)
        else dato.nombre := valoralto;
    end;

  procedure minimo (var r1,r2,r3:alumno; var min:alumno);
    begin
      if (r1.nombre<r2.nombre)and(r1.nombre<r3.nombre)
        then begin
          min := r1;
          leer(det1,r1)
        end
        else if (r2.nombre<r3.nombre)
                then begin
                  min := r2;
                  leer(det2,r2)
                end
                else begin
                  min := r3;
                  leer(det3,r3)
                end;
    end;

  begin
    assign (det1, 'det1');
    assign (det2, 'det2');
    assign (det3, 'det3');
    assign (maestro, 'maestro');

    {proceso principal}
    rewrite (maestro);
    reset (det1);
    reset (det2);
    reset (det3);

    leer(det1, regd1);
    leer(det2, regd2);
    leer(det3, regd3);
    minimo(regd1, regd2, regd3, min);

    { se procesan los tres archivos }
    while (min.nombre <> valoralto) do
      begin
        write (maestro,min);
        minimo(regd1,regd2,regd3,min);
      end;
    close (maestro);

    { ver resultados }
    reset (maestro);
    while (not eof(maestro)) do
      begin;
        read (maestro,min);
        writeln (min.nombre,min.dni);
      end;
  end.
```

Ejemplo 12.12

Los vendedores de cierto comercio asientan las ventas realizadas sobre tres computadoras, entre otros datos de interés, por cada máquina se genera un archivo donde por cada venta se registra el código del vendedor, el producto vendido y el monto total de la venta. Se sabe, además, que los vendedores pueden registrar las ventas sobre cualquiera de las computadoras y que la firma tiene un gran número de empleados. Al finalizar el día y antes de apagar cada computadora un proceso determinado ordena los archivos de vendedores por un código. Luego estos tres archivos son enviados al departamento de personal donde deben ser diariamente agrupados, dado que los empleados cobran comisión por ventas realizadas.
El siguiente algoritmo plantea el problema de la reunión de esto tres archivos sobre un archivo maestro, el cual resume para cada empleado el total de las ventas registradas en un determinado día.

Precondiciones:

- Los tres archivos recibidos están ordenados por código de vendedor.
- Cada vendedor puede realizar varias ventas diarias.

Poscondición:

- Se genera el archivo maestro de resumen con un registro por empleado.

```
program union_de_archivos_II;

  const valoralto = '9999';
  type str4 = string[4];
       str10 = string[10];
       vendedor = record
         cod: str4;                  { código de vendedor }
         producto: str10;            { descripción del producto }
         montoVenta: real;           { monto de la venta }
       end;

       ventas = record
         cod: str4;                  {código de vendedor}
         total: real;  {monto de la total para el vendedor}
       end;

       detalle = file of vendedor; { archivo que contiene
                                     información de ventas }
       maestro = file of ventas;   { archivo que contiene información
                                     de ventas totales por vendedor }

  var min, regd1, regd2, regd3: vendedor;
      det1, det2, det3: detalle;
      mae1: maestro;
      regm: ventas;
      aux: str4;
```



```
procedure leer (var archivo:detalle; var dato:vendedor);
  begin
    if (not eof( archivo ))
      then read (archivo, dato)
      else dato.cod := valoralto;
  end;

procedure minimo (var r1,r2,r3:vendedor; var min:vendedor);
  begin
    if (r1.cod <= r2.cod) and (r1.cod <= r3.cod)
      then begin
        min := r1;
        leer(det1,r1)
      end
      else if (r2.cod <= r3.cod)
             then begin
               min := r2;
               leer(det2,r2)
             end
             else begin
               min := r3;
               leer(det3,r3)
             end;
  end;

begin
  assign (det1, 'det1');
  assign (det2, 'det2');
  assign (det3, 'det3');
  assign (mae1, 'maestro');

  reset (det1);
  reset (det2);
  reset (det3);
  rewrite (mae1);

  leer (det1, regd1);
  leer (det2, regd2);
  leer (det3, regd3);
  minimo (regd1, regd2, regd3, min);

  { se procesan los archivos de detalles }
  while (min.cod <> valoralto) do
    begin
      aux := min.cod;
      {se asignan los valores para el registro del archivo maestro}
      regm.cod := min.cod;
      regm.total := 0;

      { se procesan todos los registros de un mismo vendedor }
```

```
        while (aux = min.cod ) do
          begin
            regm.total := regm.total+ min.montoVenta;
            minimo (regd1, regd2, regd3, min);
          end;

          { se guarda en el archivo maestro}
        write(mae1, regm);
      end;

    {ver resultados}
    reset (mae1);
    while (not eof( mae1 )) do
      begin
        read (mae1,regm);
        writeln (regm.cod,regm.total);
      end;
  end.
```

Nótese que la diferencia fundamental respecto del problema anterior radica en que, ahora, el nuevo archivo debe resumir uno o varios registros de los archivos recibidos.

12.4 Eliminar elementos de un archivo

Cuando se diseña un archivo se debe considerar, además, los tipos de cambios que probablemente tendrán lugar en la vida del mismo. En particular, la eliminación de archivos debe ser contemplada con sumo cuidado.

Cuando un registro deja de tener información significativa, el mismo debe ser eliminado. Para quitar un elemento se puede proceder de dos maneras posibles:

- Mediante un **borrado físico**.
- Mediante un **borrado lógico**.

12.4.1 Borrado físico

Consiste en eliminar efectivamente el elemento del archivo recuperando el espacio que el mismo ocupa. Esta opción tiene la ventaja que el archivo ocupa en todo momento el espacio estrictamente necesario en el disco rígido. Por el contrario, presenta una desventaja muy significativa; ante el borrado de cada elemento se debe efectuar un "corrimiento" de los elementos posteriores al que se desea quitar. Este método resulta muy ineficiente desde le punto de vista del tiempo de procesamiento, si bien es eficiente desde el punto de vista de la utilización de espacio de disco.

12.4.2 Borrado lógico

Consiste en indicar que la zona que ocupa el registro que se quiere eliminar, se encuentra disponible para que cualquier otro elemento la ocupe. La ventaja del método es su velocidad de procesamiento, solo hay que indicar que zona del archivo queda libre. La desventaja radica en que el espacio sobre el disco rígido no se recupera, esto es, el espacio sólo puede ser reutilizado por el mismo archivo.

Una variante del borrado lógico permite "recuperar" la información, si luego de su borrado se detecta que aún era necesaria. Basta con indicar que la zona del archivo ocupada por el registro está libre, en caso de necesitarse nuevamente la información alcanza con sacar dicha marca. Obviamente, el espacio marcado como libre puede ser reutilizado, en ese caso la información anterior se pierde definitivamente.

12.4.3 Estrategia de trabajo

Si bien no es objetivo de este libro analizar en detalle los problemas subyacentes con la eliminación de registros en un archivo, se puede concluir que la técnica de borrado lógico supera en eficiencia a la técnica de borrado físico.

Ahora bien, el proceso de borrado lógico puede generar archivos muy grandes y que, en realidad, solamente tienen un bajo porcentaje de información útil. El resto son lugares "vacíos" donde podrían agregarse elementos. Ante esta situación, es posible que periódicamente, se efectúe un borrado físico, recuperando así el espacio no utilizado. La periodicidad de este proceso depende de la política de trabajo que se establece.

La información es un bien costoso, el espacio en disco resulta cada vez más barato. Por esto, borrar información física, en la actualidad, está limitada exclusivamente a algún dato temporal mal ingresado.

12.5 Nociones generales sobre bases de datos

12.5.1 Archivos secuenciales indizados

Una alternativa posible de trabajo con archivos los constituyen los archivos secuenciales indizados. Las últimas páginas de un libro contienen, normalmente, un índice, una tabla que contiene una lista de temas y los números de página en donde pueden encontrarse dichos temas.

Definición

Se define un índice como una estructura (clave, referencia) que permite encontrar o distinguir en forma más rápida un elemento dentro de un contexto. El elemento es reconocido univocamente mediante la clave y la referencia indica la posición o lugar que ocupa.

Cuando se trabaja con un archivo de datos es posible, normalmente, establecer un identificador que permita individualizar cada uno de los elementos del archivo. Este identificador (que puede, por ejemplo, estar compuesto por uno o varios campos de un registro) es conocido con el nombre de clave o llave.

Cuando se estudió el concepto de eficiencia de algoritmos, en el capítulo 8, se observó que uno de los parámetros que afecta su comportamiento es la velocidad con que un algoritmo retorna un resultado. Estudiando este concepto con archivos, suponga que debe localizar un registro particular dentro de un archivo de alumnos, el cual está organizado de una forma secuencial; a fines prácticos se supone que los *buffers* de entrada y de salida relacionados solo tienen capacidad para almacenar un registro. El mejor caso consiste en tener disponible en el *buffer* de entrada el registro que se intenta recuperar; por el contrario, puede ocurrir que el registro deseado sea el último que se recupere, en ese caso y suponiendo que el archivo tiene N elementos, se necesitan N accesos o lecturas (recordar que los *buffers* tienen, hipotéticamente, capacidad para un registro). El caso promedio será, pues N/2 accesos al disco rígido.

Si se dispone de un índice asociado al archivo, la búsqueda del elemento se realiza sobre dicho índice. Suponga que dicho índice está almacenado en una estructura auxiliar en memoria RAM, localizar el registro requiere un tiempo de procesamiento mucho menor al no requerir accesos a disco. Una vez localizado el registro en la estructura auxiliar, se accede a la posición del archivo que la misma indica, en forma directa. Si se toma como ejemplo Pascal, se puede utilizar la instrucción seek, pero el lector debe notar que esta instrucción no permite acceder directamente a la posición deseada. El acceso que provee es al elemento que se le indica, obtenido en forma secuencial desde el comienzo del archivo. Por lo tanto para poder acceder directamente a un elemento del archivo no solo es necesaria una estructura de índices auxiliar, sino la posibilidad del lenguaje para realizar ese acceso directo a un elemento.

Ejemplo 12.13

Los vendedores de un comercio registran las ventas realizadas en un archivo maestro, donde por cada venta se guarda el código del vendedor, el producto vendido y el monto total de la venta. Al finalizar el día, se desea obtener dicha información ordenada por código de vendedor. El programa presenta una solución al problema.

```
program archivos_indizados;
 const cantidadregistros = 100;
 type  str4 = string[4];
       str10 = string[10];
       vendedor = record
          cod: str4;                  { codigo de vendedor }
          producto: str10;            { descripcion del producto }
          preciounitario: real;       { precio unitario del producto }
       end;
       datos = record
          nreg: integer;
          clave: str4;
       end;
       detalle = file of vendedor; {archivo que contiene}
                                   {información diaria}
```



```
       indice = array[ 1..cantidadregistros ] of datos;
    var regd: vendedor;
        det1: detalle;
        ind: indice;
        i, j: integer;
        codigo: str4;
  procedure ordena( var a: indice; n: integer; campo: str4 );
    var i, j: integer;
        item: datos;
        ok: boolean;
    begin
      for i := 2 to n do
       begin
         item := a[i];
         j := i;
         ok := false;
         while ( j > 1 ) and ( not ok ) do
          if ( a[ j-1 ].clave <= item.clave )
              then ok := true
              else begin
                 a[ j ] := a[ j-1 ];
                 j := j - 1;
              end;
          a[ j ] := item;
       end;
    end;
    begin
      assign( det1, 'detalle.dat' );
      reset( det1 );  {recorre el archivo guardando en el arreglo
                       el campo clave y la posición en el archivo
                       del registro actual }
        i := 0;
        while ( not eof( det1 ) ) do
          begin
            read( det1, regd );
            i := i + 1;
            ind[ i ].nreg := i;
            ind[ i ].clave := regd.cod;
          end;
      { ordena el vector generado de acuerdo al campo clave }
      ordena( ind, i, codigo );
      { recorre el archivo de acuerdo al orden que presenta
        el vector ind }
      reset( det1 );
      for j := 1 to i do
          begin
            seek( det1, ind[ j ].nreg - 1 );
            read( det1, regd );
            writeln( regd.cod, regd.producto, regd.preciounitario );
          end;
    end.
```

Nótese que la organización adecuada para los archivos indizados, en general, consiste en representar la información mediante una estructura arbórea, generando lo que se denomina un **árbol balanceado**.

Si se desea utilizar una estructura de memoria auxiliar para almacenar el índice, se presentan varios inconvenientes. El más importante radica en que esta estructura debe ser compartida por todos los usuarios del archivo. Esto presenta soluciones muy difíciles o imposibles de implementar para el problema. Se sugiere al lector profundizar el tema con la bibliografía que se indica en el siguiente apartado.

12.5.2 Bases de datos

Las limitaciones de los sistemas orientados a archivos puramente secuenciales no los privaron de ser herramientas eficaces para producir algoritmos de liquidación de sueldos, facturas y emisión de otros informes generales durante cierto tiempo. Sin embargo, para ejecutar muchas tareas rutinarias en los negocios se necesitan características que este tipo de archivos no provee. El acceso directo a los datos (la capacidad de tener acceso y procesar directamente un registro dado sin ordenar el archivo o leer los registros en secuencia) o la capacidad de compartir estructuras auxiliares (árboles balanceados de búsqueda) entre diferentes usuarios son ejemplos de estas situaciones (Folk, 1992), (Smith, 1988).

Los sistemas de archivos presentan una serie de inconvenientes como redundancia de los datos, pobre control sobre los mismos, capacidades limitadas de manejo y esfuerzos excesivos de programación (Date, 2001). Los sistemas de bases de datos superan estas limitaciones.

Definición

Una base de datos es un conjunto autodescriptivo de registros integrados. Es autodescriptivo porque, además de los datos fuente del usuario, contiene una descripción de su propia estructura (diccionario de datos). Una base de datos incluye archivos de datos del usuario, índices que se usan para representar las relaciones entre los datos y para mejorar el desempeño de las aplicaciones de la base de datos, aplicaciones que utilizan los datos, estructuras de entrada de datos y reportes, etc. (Kroenke, 1996).

Definición

Una base de datos es una colección de elementos de datos interrelacionados que pueden ser accedidos en forma compartida por un gran número de computadoras (Hansen, 1997).

No es objetivo de este libro desarrollar más en detalle el concepto de bases de datos, para ello se sugieren, además de las anteriores, las siguientes referencias (Ramakrishnan, 2000), (Ullman, 1999), (Batini, 1994), (Elmasri, 2000).

Conclusiones

Si bien desde un punto de vista algorítmico el tema de archivos no introduce novedad para el lector, la importancia en las aplicaciones reales del tratamiento de información almacenada en memoria auxiliar impone su tratamiento.

En este capítulo se han presentado los problemas básicos de creación, lectura, modificación y crecimiento de archivos utilizando la sintaxis de Pascal. Asimismo, se introduce la definición de base de datos, tratando que el lector asocie la misma con una evolución al concepto de archivos.

Ejercicios

1. a) Generar un archivo conteniendo los datos personales de los clientes de un comercio. Estos datos son: nombre, dirección, teléfono y tope de crédito.
   b) Realizar un proceso que reciba el archivo anteriormente generado e imprima su contenido.
   c) Actualizar el archivo anterior aumentando en un 20% el tope de crédito.

2. Realizar un proceso que reciba un archivo de clientes de cierta empresa, con el código de cliente, nombre y saldo; y otro archivo con el código de cliente y monto de la operación, y realice el proceso de actualización del archivo de clientes.
   a) suponiendo que cada registro del archivo de clientes tiene cero o una modificación
   b) suponiendo que cada registro del archivo de clientes tiene cero, una o más modificaciones
   c) suponga, ahora, que pueden existir tres archivos detalles.

3. Una empresa tiene información sobre las ventas realizadas por los empleados con la siguiente estructura: nombre de sucursal, nombre de empleado, fecha de la operación y monto de la misma. El archivo está ordenado por sucursal, empleado y fecha, pudiendo existir varios registros para un empleado en una determinada fecha. Por cuestiones estadísticas se desea obtener un listado donde figure el monto facturado por cada empleado en cada día, el monto total por empleado y el monto total por sucursal.

4. Tres sucursales de una empresa generan, cada una, un archivo de ventas realizadas. Dicho archivo tiene la siguiente estructura: producto vendido, cantidad vendida. Dicho archivo es enviado, desde cada sucursal, a casa central donde es procesado en conjunto con los demás archivos de detalle recibidos. En casa central se genera un archivo resumen donde por cada producto figura la cantidad vendida. Realizar dicho proceso teniendo en cuenta que el archivo de cada sucursal está ordenado por producto vendido y que pueden existir varios registros con el mismo producto.

13

Capítulo 13

Conceptos de metodologías en el desarollo de sistemas de software

13

Conceptos de metodologías en el desarrollo de sistemas de software

Objetivo

A lo largo del texto se han discutido conceptos relacionados con la programación de algoritmos. En particular hemos atacado lo que se conoce como *programming in the small*, es decir la expresión simbólica detallada y verificable de algoritmos. En este capítulo se sitúan los conceptos estudiados en el contexto más amplio de software y de Ingeniería de Software.

El eje temático del capítulo es la ubicación del proceso de análisis y expresión de algoritmos dentro de lo que se llama ciclo de vida del software en el desarrollo de sistemas. Interesa particularmente reflexionar con el lector sobre el conjunto de conocimientos que representa la Ingeniería de Software (en particular aspectos de *programming in the large*) y dentro de ellos cuáles son los aspectos que se han cubierto hasta ahora en el texto.

En particular se discute conceptualmente la noción de *metodología* de análisis y diseño de software y, generalizando la discusión de la metodología de refinamientos sucesivos con un lenguaje procedural utilizada en el texto, se menciona la noción de paradigma de programación.

La intención del capítulo es mostrar que los conceptos analizados en los caítulos anteriores del texto constituyen un núcleo básico sobre el cual se abre al lector el enorme desafío que representa la Ingeniería de Software.

13.1 Ingeniería de Software

Se pueden analizar numerosas definiciones de **Ingeniería de Software**, varias de las cuales se transcriben en el Apéndice B. En este capítulo se considera la que presenta Ghezzi en Ghezzi,1991:

> **Definición**
>
> La Ingeniería de Software es el área de la ciencia Informática que trata el análisis, diseño e implementación de sistemas de software.

Se entiende por sistema de software un conjunto de programas relacionados, orientados a resolver un problema del mundo real que, por su complejidad y tamaño, requieren para su solución un equipo de especialistas en software.

A diferencia de la programación de una aplicación que normalmente es una tarea individual, la Ingeniería de Software debe verse como un desarrollo en equipo, donde los programas diseñados e implementados por un especialista pueden ser verificados y/o modificados por otro. Más aún, al pensar en un sistema de software normalmente se concibe su aplicación durante un tiempo prolongado, con lo que posteriores modificaciones, correcciones o ampliaciones al sistema pueden ser realizadas por un equipo de especialistas diferente del original.

La noción de **Ingeniería** debe asociarse con la existencia de una metodología de **producción** de software, fundamentada en principios establecidos y utilizando técnicas probadas en la práctica (piense el lector en una Ingeniería clásica como la de construcción de edificios, puentes y caminos con cientos de años de experiencia).

Sin embargo, la Informática ha tenido una evolución tan acelerada en tan corto tiempo que se puede afirmar que el proceso de establecimiento de los fundamentos, métodos y herramientas de desarrollo de sus productos está aún en maduración.

13.2 El ciclo de vida clásico de un sistema de software

La Figura 13.1 muestra el esquema clásico de las etapas en la vida de un sistema de software. Por su estilo secuencial, esta división en etapas se suele llamar "ciclo de vida en cascada".

En este libro sólo se discutirá el significado de cada una de las etapas, sin analizar críticamente la descripción del ciclo de vida del software que ofrece el modelo en cascada. De todos modos debe aceptar el lector que existen modelos alternativos de ciclo de vida y que en la realidad no existe normalmente una separación tan nítida entre las etapas.

13.2.1 Análisis y especificación de requerimientos

Normalmente es la primera etapa en el desarrollo de un sistema de software.

El objetivo primordial es relevar, documentar y especificar los requerimientos del sistema de software, teniendo especialmente en cuenta el contexto y el enfoque del usuario.

Esto significa una intensa tarea de análisis del mundo real en el que se desenvuelve la actividad del usuario, donde el sistema de software se ha de insertar (por ejemplo, la administración de la empresa donde se decide implantar una gestión computarizada), cuyo resultado final debiera ser un detalle formalizado de los objetivos, prioridades y restricciones del sistema de software a desarrollar.

Normalmente, desde el punto de vista de la organización que desarrolla el software, esta etapa debiera incluir los estudios de factibilidad y riesgo del proyecto así como el análisis de costo del sistema.

13.2.2 Diseño y especificación del sistema

La etapa de análisis de requerimientos establece el **qué** se debe hacer con el sistema. A continuación debe definirse el **cómo** llegar a dichos objetivos. Para esto se suele trabajar en dos niveles de diseño del sistema de software:

- La concepción de la arquitectura del sistema, que incluye la definición del soporte físico de contexto y la subdivisión funcional del sistema en módulos.
- El diseño detallado que refina la especificación de cada uno de los módulos y su interfaz de comunicación intra-módulos y con el usuario.

Así como la documentación final del análisis de requerimientos se aproxima a un texto riguroso que describe contexto, expectativas, restricciones y objetivos, la documentación del diseño puede utilizar lenguajes de especificación que combinen elementos gráficos y formales aproximándose a una codificación de alto nivel.

13.2.3 Codificación y verificación de los módulos

La producción efectiva del código de cada uno de los módulos y la prueba funcional de los mismos con datos que representen la operatoria normal del sistema constituyen el núcleo de esta etapa.

La tarea tradicional de programación y puesta a punto es muy importante para el desarrollo del sistema y dependerá fuertemente de la calidad y rigurosidad del diseño elaborado en la etapa anterior. Más aún, en la medida en que se trate de incrementar la productividad automatizando la generación de código, será fundamental partir de especificaciones de diseño unívocas y que sean sintáctica y semánticamente correctas.

13.2.4 Integración y prueba global del Sistema

Los módulos desarrollados y probados independientemente (por ejemplo, la emisión de remitos y facturas o el subsistema de bancos o el módulo de control de *stock* en un negocio cualquiera) deben acoplarse en un sistema único y realizar una prueba funcional (dinámica) con el sistema en su ambiente real, a fin de llegar a la entrega definitiva del sistema al usuario.

Es muy importante la relación de esta prueba global con el documento de requerimientos inicial. Normalmente en el documento de requerimientos se define la metodología para probar que el sistema desarrollado satisface los objetivos del usuario. Por ello se suele decir que se establecen condiciones contractuales que serán chequeadas en el momento de la prueba global del sistema.

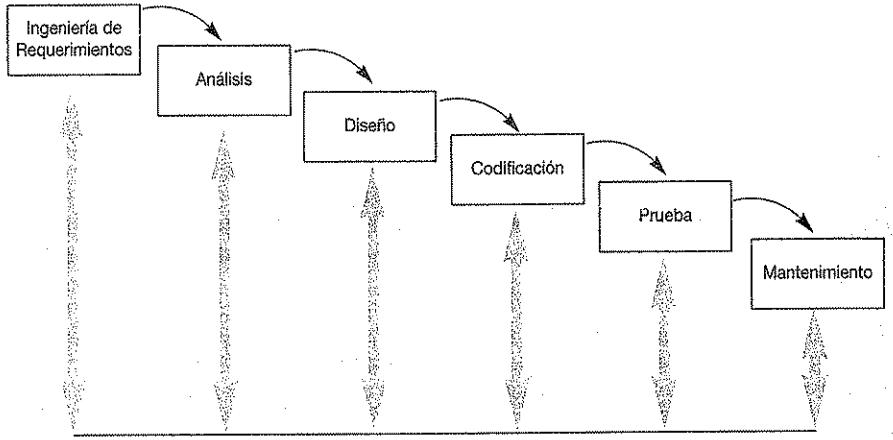


Figura 13.1

13.2.5 Mantenimiento del Sistema

Un concepto ideal de calidad en un sistema de software lleva a suponer que, una vez desarrollado e instalado, el sistema debería ser **estable y libre de errores**. Esto significaría que la organización que desarrolla el software ha entregado un producto definitivo y que su tarea prácticamente ha concluido... pero esta hipótesis ideal solo se da en un ínfimo porcentaje de los sistemas de software.

En la realidad, una vez entregado el sistema de software comienza la etapa tal vez más compleja de todo el ciclo de vida: normalmente los sistemas fallan, se modifican y crecen. Los tres aspectos (errores en la concepción y/o el desarrollo del sistema, modificaciones a pedido del usuario y crecimiento de las especificaciones iniciales) producen un impacto importante en las prestaciones y en el costo de funcionamiento del sistema.

La calidad de la metodología y herramientas empleadas para diseñar y desarrollar el software condicionará fuertemente la facilidad y el costo de mantenimiento del sistema.

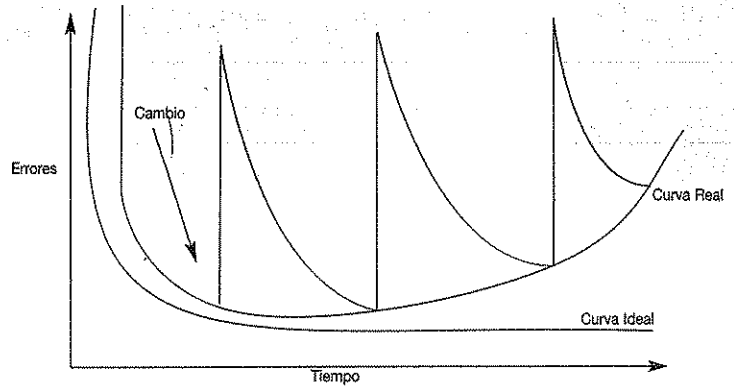


Figura 13.2

13.2.6 Tiempos y costos requeridos por las etapas del ciclo de vida

En los textos clásicos de Ingeniería de Software el lector encontrará diferentes análisis de los costos relativos de cada etapa del ciclo de vida de un sistema de software, incluyendo o no la etapa de mantenimiento en el análisis. Esto se puede apreciar en las Figuras del Apendice E.

Si bien cualquier análisis de los costos relativos de cada una de las etapas del ciclo de vida de un sistema de software depende fuertemente del tipo de sistema, es decir, de su área de aplicación, se pueden hacer algunos comentarios generales que el lector experimentado posiblemente haya apreciado en su vida profesional:

- Los errores y modificaciones se hacen más costosos a medida que estamos más avanzados en el desarrollo de un sistema. De hecho un error importante en la etapa de mantenimiento puede obligar a replantear incluso el documento de requerimientos y consecuentemente a redefinir el análisis, el diseño y la codificación del sistema.
- En la etapa tradicional de programación y prueba, difícilmente se invierte más del 15% del tiempo y costo de desarrollo. Asimismo es la etapa más fácilmente automatizable, con lo que la mano de obra es menos especializada y, por ende, de menor costo.
- La actividad creativa es posiblemente el diseño, donde el especialista en software opta por alternativas de acuerdo a sus conocimientos. Se estima en un 25% del costo y esfuerzo de un proyecto de software complejo el que corresponde a esta etapa.
- Es imprescindible contar con equipos multidisciplinarios y con gran capacidad de comprensión del ambiente y actividad del usuario en la etapa de análisis de requerimientos.

El porcentaje del tiempo y costo del desarrollo del proyecto que se invierten en esta etapa son críticos para disminuir el número y costo de los errores y modificaciones del sistema en el futuro.

- El usuario debe participar efectivamente tanto de la especificación de requerimientos como de las pruebas parciales y finales del sistema. Su intervención permite ajustar el modelo desarrollado al contexto real mucho más rápidamente. Por otra parte, si el usuario debe cambiar algún aspecto de su trabajo, se adaptará más fácilmente al cambio si se siente partícipe de la concepción y prueba del sistema.
- La etapa de mantenimiento de los sistemas de software es costosa y de costo creciente en el tiempo. De hecho los especialistas estiman que al menos el 35 % del esfuerzo y costo del sistema está contenido en esta etapa. Una simple mención del impacto y costo que puede tener una modificación luego de un largo período de funcionamiento de un sistema la puede dar la corrección que se debió efectuar para tener en cuenta el cambio de centuria que se dio en el año 2.000, lo que impactó sobre todo el software que fuera concebido con un manejo de fechas relativo al 1.900. En los últimos dos años la hora de programador "de mantenimiento" orientado a corregir sistemas con estos problemas se ha incrementado más del 80%.

13.3 Especificación, codificación y prueba de algoritmos

A lo largo del texto se han desarrollado conceptos relacionados con la especificación, codificación y prueba de algoritmos. Normalmente este núcleo esencial de la actividad en el desarrollo de software se denomina *programming in the small*.

Si bien el costo y tiempo de la actividad de codificación y prueba es relativamente pequeña dentro del desarrollo de un sistema complejo de software, es fundamental para el especialista en Informática comprender sus alcances y el modo correcto de desarrollar programas **eficientes y verificables.** Para verificar corrección deberá chequear el cumplimiento de la especificación realizada *a priori*. Por esto el programador debe tener presente en todo momento lo especificado para cada módulo de software a desarrollar.

En el desarrollo de la Informática, a partir de especificaciones unívocas y correctas, se está haciendo un gran esfuerzo por lograr la automatización en la generación del código que responde a la especificación. Normalmente es difícil lograr una generación de código muy eficiente, pero en compensación la automatización incrementa la productividad de la organización que desarrolla el software (entendiendo como medida de la productividad el número de líneas de código correctas producidas por unidad de tiempo).

De todos modos, la elección de alternativas algorítmicas para una misma función o el perfeccionamiento de un algoritmo determinado sigue siendo una tarea que requiere la creatividad humana.

13.4 El método de refinamientos sucesivos

También se ha insistido en el texto en una metodología que va **de lo general** (concepción del sistema, análisis del algoritmo global, definición de las estructuras de datos necesarias) **a lo particular** (desarrollo de cada uno de los módulos que componen el sistema, ajuste de cada uno de los algoritmos, restricciones en los datos).

Esta metodología se llama de refinamientos sucesivos:

- Primero se modeliza el problema del mundo real.
- Se analizan la estructura global del sistema y los datos que representan el problema.
- Se descompone la solución en partes (modularización).
- Se desarrollan cada uno de los módulos básicos y se los prueba.
- Si es necesario se sigue refinando cada módulo en sus partes elementales.

Desde el punto de vista formativo, el criterio de abstraer lo general y derivar lo particular es muy útil.

Sin embargo, en la concepción de software de alta productividad (trabajando sobre el concepto de reusabilidad) muchas veces se utilizan módulos previamente desarrollados con una función idéntica o más general que se adaptan al problema que se quiere solucionar. En este caso no se trabaja estrictamente por refinamientos sucesivos, sino que total o parcialmente se adopta una técnica *bottom up* o de integración de soluciones parciales previamente desarrolladas.

13.5 La noción de paradigma de programación

A lo largo del texto se ha puesto énfasis en la codificación de algoritmos mediante instrucciones que se ejecutan secuencialmente, siguiendo una línea de control única. Este modo de escribir algoritmos (que parece natural) se denomina procedural o imperativo. El paradigma de programación imperativo asume una arquitectura de ejecución de los algoritmos que es capaz de seleccionar la instrucción siguiente, interpretarla y ejecutarla. Tal modelo de arquitectura se denomina Von Neumann en relación con el esquema clásico de las computadoras digitales (Prieto,1995), (Hennessy,1993).

Sin embargo, el lector debe saber que existen otros modelos de arquitectura física diferentes del esquema de Von Neumann (basta con pensar en los sistemas en red o los procesadores paralelos, por ejemplo) y también otros paradigmas de programación.

Cada paradigma de programación significa un modo diferente de analizar y especificar los algoritmos y clases de lenguajes de programación distintos (adecuados al estilo de especificación).

Normalmente se estudian al menos dos paradigmas notoriamente diferentes del modelo imperativo, que son la programación funcional y la programación lógica (Bird,1988).

Por otra parte, el énfasis puesto en el modelo de objetos en la descomposición de problemas del mundo real y en los recursos de abstracción de datos (tal como hemos visto en el capítulo 9) han dado lugar al desarrollo de algoritmos con orientación a objetos. Si bien es discutible hablar de un paradigma de programación orientado a objetos, el lector seguramente apreciará que el enfoque de objetos constituye uno de los modelos más importantes en el desarrollo de sistemas de software actual (Wirfs,1990), (Martin,1994).

En el capítulo 14 se discutirá la programación orientada a eventos y los lenguajes visuales, con el objetivo de dar al lector una visión diferente del modelo de desarrollo de software expuesto hasta el momento.

13.6 El rol del especialista en software

Más allá de la metodología empleada y las técnicas o recursos elegidos (paradigma y lenguaje de programación, herramientas de especificación, técnicas de documentación, métodos de prueba de algoritmos, etc) el especialista en software trabaja fundamentalmente en las etapas de la Ingeniería de Software.

Entonces (recordando la definición que se ha dado de Ingeniería de Software) debe estar capacitado para:

Definición

- **Analizar problemas del mundo real, abstrayendo sus requerimientos.**
- **Diseñar la posible solución de los mismos con herramientas informáticas.**
- **Establecer métodos de verificación y prueba de los sistemas de software desarrollados.**
- **Mantener los sistemas asimilando las modificaciones y ampliaciones solicitadas por el usuario.**
- **Y, sobre, todo tener capacidad de elección entre los recursos que ofrece la tecnología (de hardware y software) que sean los mejores para el problema a resolver.**

El cambio tecnológico que impone la Informática obliga al especialista de software a no ceñirse a métodos o recursos fijos. Ante cada problema debe tener capacidad para analizar y elegir la herramienta informática más adecuada para su solución.

Otro aspecto esencial es que la Informática tiene aplicaciones multidisciplinarias, lo que obliga al especialista de software a estudiar problemas del mundo real en áreas muy diferentes, tratando de comprender el contexto del usuario.

13.7 Aspectos importantes de los sistemas de software

La Ingeniería de Software (como todas las Ingenierías) debe elaborar productos que son sistemas de software.

Como producto, los sistemas de software tienen una característica complicada que es el cambio que pueden sufrir a lo largo del tiempo. Este cambio (modificación, crecimiento) provoca un gran impacto en el mantenimiento del producto software (ver Figura 13.2).

Por otra parte, como en el análisis de cualquier producto, debe ponerse énfasis en la calidad del software.

Esta calidad es más difícil de medir que la de un producto convencional (pensar en un automóvil como producto: se puede medir la *perfomance* de su motor, la calidad de su pintura, la capacidad del baúl, el diseño interior, etc.), pero se discutirán ocho aspectos importantes del software que hacen a la calidad del mismo:

13.7.1 Corrección

Ante todo, interesa poder establecer una equivalencia matemática entre la especificación y el funcionamiento del software. La corrección funcional de un sistema de software significa que responde exactamente a lo especificado.

13.7.2 Verificabilidad

Para poder demostrar que un software es correcto, se debe realizar una prueba del mismo. El desarrollo de sistemas de software verificables se basa en la utilización de metodologías y técnicas adecuadas (documentación, modularización, definición correcta de los datos, etc). Naturalmente es un atributo que hace a la calidad interna del producto, pero también puede ser exigida por el usuario que debe probar determinadas condiciones de funcionamiento.

13.7.3 Confiabilidad

Como en muchos otros productos, interesa medir el tiempo medio entre fallas (MTFB) de un sistema de software. Naturalmente a mayor MTFB se dice que el sistema es más confiable y este es un atributo de calidad muy importante para el usuario. La corrección funcional en condiciones normales de un sistema de software no asegura que a lo largo de su vida útil no aparezcan errores derivados de cambios en las condiciones de contexto. Por otra parte, las modificaciones y ampliaciones de los sistemas de software generalmente están acompañadas de períodos de crecimiento de las fallas de funcionamiento (Pressman,1998).

13.7.4 Eficiencia

Un atributo de calidad importante es la eficiencia en el uso de los recursos por cada uno de los módulos del sistema de software. Normalmente la eficiencia de los algoritmos (tal como vimos en el capítulo 8) se mide por el tiempo de ejecución de los mismos y por la memoria que requieren. Por otra parte, hay aplicaciones donde la eficiencia es crítica para satisfacer los reque-

rimientos del usuario (piense el lector en el software de un cajero bancario o en el software de un piloto automático de avión que se desplaza a 1.000 Km/hora).

13.7.5 Interfaz amigable

Normalmente los sistemas de software son utilizados por seres humanos. Por ello es fundamental para la calidad del sistema que el modo de comunicación entre el sistema y el usuario sea "amigable", es decir, fácil de comprender y utilizar. Muchas veces este atributo de presentación hace elegible un producto de software frente a otras alternativas.

13.7.6 Facilidad de mantenimiento

Se ha visto largamente la importancia de la etapa de mantenimiento en los sistemas de software de cierta importancia. Por ello, la utilización de técnicas de análisis y diseño del software que lo hagan claro, fácil de comprender y de modificar representan un atributo de calidad esencial para el costo global del sistema de software.

Cuando se habla de mantenimiento (Yourdon,1989) se tiene no solo el aspecto correctivo de posibles fallas, sino el mantenimiento adaptivo y perfectivo relacionado con nuevos requerimientos del usuario sobre un sistema que ya está operativo.

La facilidad de mantenimiento se mide en el tiempo que requiere de la organización de software el ajuste del sistema a modificaciones y cambios.

13.7.7 Reusabilidad

Desde el punto de vista de la Ingeniería de Software, la reusabilidad del sistema es un atributo esencial de calidad. El grado de generalidad de la solución y su adaptabilidad para el reuso con nuevos usuarios, tiene un impacto fundamental en la rentabilidad de la organización que desarrolla sistemas de software.

Por esto, una línea importante de investigación en Informática en los últimos diez años ha sido centrar el proceso de desarrollo de sistemas en la idea del reuso. La orientación a objetos y todas sus derivaciones (Biggerstaff,1984), (Liskov,1974), (Stepoway,1987) hacen hincapié esencial en la reusabilidad de los sistemas.

13.7.8 Portabilidad

Asociada con la reusabilidad está la noción de portabilidad de los sistemas de software. Los sistemas portables son aquellos que se adaptan a los cambios de contexto (diferentes sistemas operativos, diferentes arquitecturas de máquinas, diferentes características de las máquinas, etc.) con mínimos cambios en el software. Como en el caso de la reusabilidad, la portabilidad exige un mayor esfuerzo de desarrollo, pero incrementa la rentabilidad de la organización de software en el caso de tener que adaptar el sistema a nuevos ambientes de trabajo (Johnson,1978).

13.8 Principios generales de la Ingeniería de Software

Completando las ideas sobre Ingeniería de Software que permitan al lector ubicar la programación en el contexto más general de la Informática, se mencionan los principios generales de la Ingeniería de Software siguiendo los lineamientos de Ghezzi (Ghezzi,1991):

- La **abstracción** es el recurso fundamental del ingeniero de software para controlar y reducir la complejidad del mundo real.
- La capacidad de **separación de objetivos** permite analizar y descomponer el problema real, aislando sus funciones. De este modo se puede modularizar y desarrollar simultáneamente soluciones para cada objetivo.
- La **especificación formal** de los problemas es un principio esencial para lograr verificar la corrección de las soluciones.
- **Modularizar** soluciones es un principio propio de las Ingenierías (*divide and conquer*) La modularización simplifica el desarrollo y el mantenimiento de los sistemas.
- Ante cada problema es muy importante descubrir las características del mismo que permitan que su solución sea reusada en un contexto más grande de problemas. Este principio se denomina **generalización** y ayuda a la productividad del software.
- El principio de **desarrollo incremental** significa que a partir de una solución sobre los aspectos básicos del problema del mundo real, se puede refinar agregando aspectos que potencien o completen la solución. Siempre es mejor poner a punto un prototipo del sistema y a partir de él refinar un nuevo prototipo que se acerca a la solución definitiva.
- Por último, un principio muy importante de la Ingeniería de Software es la **anticipación del cambio**, es decir, la capacidad de diseñar soluciones que tengan en cuenta posibles modificaciones o ampliaciones que en el futuro ha de solicitar el usuario. Esta capacidad incrementa la velocidad de respuesta y la rentabilidad de la organización de software en base a la explotación de un sistema en funcionamiento.

Conclusiones

En este capítulo se ha intentado contextualizar los conceptos de algorítmitica y estructuras de datos vistos en los capítulos anteriores, dentro del desarrollo de sistemas de software.

Se han presentado las nociones fundamentales de Ingeniería de Software, y algunos principios directores para el analisis y diseño de sistemas.

Por último, se abre al lector la perspectiva de diferentes **paradigmas de programación**, concepto que se profundizará en los capítulos siguientes.

Referencias bibliográficas

Arainty, "Madurez del Proceso y Calidad del Software", Tesis de maestría, ITBA, 1996.

Biggerstaff, Perlis, "Special Issue on Software Reusability", IEEE Trans. Software Engineering, vol SE-10, Nro 5, September 1984.

Bird, Wadler, "Introduction to Funtional Programming", CAR Hoare Series, Prentice Hall, 1988.

Bohem, "Software Engineering", IEEE Transactions on Computers, C-25, num.12, diciembre.

Bauer, "Software Engineering", North Holland Publishing Co., Amsterdam, 1972.

Fairley, "Ingeniería de Software", McGraw Hill, 1987.

Ghezzi, Jazayeri, Mandrioli. "Fundamentals of Software Engineering", Prentice Hall, 1991.

Hennessy, Patterson, "Arquitectura de Computadores. Un enfoque Cuantitativo", McGraw Hill, 1993.

IEEE Standards Collection: "Software Engineering", IEEE Standard 610.12-1990, IEEE, 1993.

Johnson, Ritchie, "Portability of C programs and the UNIX system", Bell System Technical, 1978.

Liskov, Zilles, "Programming with abstract data types", ACM Sigplan, 1974.

Martin, Odell, "Análisis y diseño orientado a objetos", Prentice Hall Hispanoaméricana, 1994.

Pressman, "Ingeniería de Software: un enfoque práctico", Mc Graw Hill, 1998.

Prieto, Lloriz, Torres, "Introducción a la Informática", Mc Graw Hill, 1995.

Stepoway, Arnold, "The Reuse System: Cataloging And Retrieval of Reusable Software", Proceeding of COMPCON 87, IEEE, 1987.

Wirfs, Brock, Wilkerson, Wienes, "Desingning Object-oriented Software", 1990.

Yourdon, "Análisis estructurado moderno", Prentice Hall, 1989.

Zelkovitz, Shaw, Gannon, "Principles of Software Engineering and Design", Prentice Hall, Englewoods Clif, 1979.

14

Capítulo 14

Programación orientada a eventos

14

Programación orientada a eventos

Objetivo

En este capítulo se discutirá la programación orientada a eventos y los lenguajes visuales, con el objetivo de dar al lector una visión diferente del modelo de desarrollo de software expuesto hasta el momento.

El eje temático del capítulo es el análisis de la ejecución de algoritmos a través de eventos y su diferenciación de la programación procedural o imperativa.

Se analizará el lenguaje y ambiente de programación Visual Da Vinci desde esta nueva óptica, y se presentarán los lenguajes visuales en general, discutiendo las características, facilidades y ventajas que el programador puede obtener de ellos.

14.1 Programación orientada a eventos

La programación procedural o imperativa, paradigma utilizado en los capítulos anteriores, constituye una de las alternativas posibles para resolver un problema. Como fue descrito anteriormente, su funcionamiento se basa en la ejecución de instrucciones siguiendo una línea de control (secuencia imperativa) única.

Asociadas con la programación imperativa se introdujeron las técnicas de análisis descendente (*top down*) y de modularización, desarrollando una metodología clásica para el desarrollo de sistemas conocida como programación estructurada con ciclo de vida en cascada (Linger Mills, Witt, 1980)

Sin embargo, tal como se explicó en el capítulo anterior, existen otros paradigmas de programación (a los que se asocian metodologías de análisis y diseño de sistemas diferentes). La programación orientada a eventos es uno de ellos (Dubois, 1995; Charte, 1998).

Su principal diferencia con el modelo procedural radica en el modo de adjudicar el control dentro de la aplicación. Mientras que en la programación imperativa, es el programador quien especifica el orden en el cual las instrucciones se ejecutan, en la programación orientada a eventos ese orden es manejado por hechos externos o **eventos**, cuya aparición no puede ser conocida *a priori*.

Algunos ejemplos de eventos son: el click hecho sobre un botón, el ingreso de un valor por parte del usuario y la elección de una opción dentro de un menú desplegable.

La selección del paradigma a utilizar se encuentra estrechamente relacionada con el tipo de problema a resolver. A su vez, esta selección determina las herramientas más útiles tales como el lenguaje, el ambiente de programación, el sistema operativo, etc.

A continuación se analizarán dos problemas concretos a fin de clarificar este concepto.

Suponiendo el siguiente problema: una empresa que fabrica torres de alta tensión está interesada en desarrollar una aplicación destinada a registrar las siguientes condiciones climáticas de una región: velocidad del viento, humedad, presión y cantidad de lluvia. A través de esta información, y luego de haber recolectado una cantidad significativa de datos, se podrán realizar estimaciones del deterioro de las torres a instalar en el lugar, transcurrido un cierto tiempo.

Como se puede advertir, la recolección de esta información es un proceso dedicado que rara vez será interrumpido por "hechos externos". La solución de este problema se basa en indicar la manera exacta en que los datos deben ser muestreados. La aplicación deberá ocuparse del control de los sensores y del almacenamiento correcto de la información. En este caso, la aplicación del paradigma procedural resulta natural.

Por otro lado, se tiene un segundo problema: una empresa de video juegos que desea desarrollar una nueva versión de un juego interactivo de fútbol para uno o dos jugadores. Independientemente de qué juego se trate, es el usuario quien conduce la evolución del mismo. No es posible escribir una solución estática, con una única línea de ejecución. Quien esté jugando debe tener la libertad de seleccionar la jugada a aplicar. En este caso, la programación orientada a eventos es el paradigma adecuado.

En este nuevo modelo, el programador debe indicar qué hacer ante la aparición de cada evento posible y serán estos hechos externos los que determinen el orden de ejecución.

En el problema anterior, es necesario implementar la ejecución de un penal, un tiro libre, etc., pero el momento en que cada uno de ellos se ejecute estará determinada por el jugador y por la evolución del juego. No es posible establecer esa secuencia de antemano.

Tal vez este sea el punto más difícil de comprender para un programador que está habituado a la programación imperativa. En el paradigma orientado a eventos no existe la figura de un programa principal que impone un orden de ejecución a los módulos que componen la aplicación, sino que se trabaja con la idea de **proyecto** dentro del cual se encuentran objetos que reaccionan frente a la ocurrencia de eventos.

A fin de facilitar la tarea del programador, los lenguajes orientados a eventos son visuales y disponen de una importante cantidad de objetos, también llamados **componentes**, que pueden ser utilizados (y re-utilizados) en la resolución del problema.

Se dice que un lenguaje es **visual** si brinda al programador la posibilidad de combinar estos componentes a través de una interfaz gráfica. Por lo general, los objetos están ubicados sobre una paleta de componentes y el programador puede seleccionar los que desee insertándolos dentro de la aplicación. El lenguaje visual es el encargado de generar (incorporar) el código correspondiente a dichas selecciones.

Sin embargo, que un lenguaje sea visual no implica que sea orientado a eventos. Dicho en otras palabras, un lenguaje puede poseer una interfaz gráfica para interactuar con el programador y no disponer de componentes que reaccionen a eventos.

Por ejemplo, el lenguaje Visual Da Vinci es visual pues, además del Editor de Código, posee un **Editor de Diagramas** que permite escribir un programa seleccionando las componentes necesarias de la paleta de componentes gráficos (LIDI, 2000).

La Figura 14.1 muestra la ventana correspondiente al Editor de Diagramas donde se ha escrito un programa que permite al robot recoger todas las flores que se encuentran en las 10 primeras calles de la avenida 1.

La parte superior de la pantalla corresponde a la paleta de componentes. En ella se encuentran representados todos los elementos necesarios para escribir un programa utilizando la sintaxis del robot. El sector inferior está dividido en tres partes: la primera es para la definición de procesos, la segunda corresponde al sector de variables y la última al desarrollo del programa principal. En este ejemplo mínimo, las dos primeras partes del sector inferior se encuentran vacías.

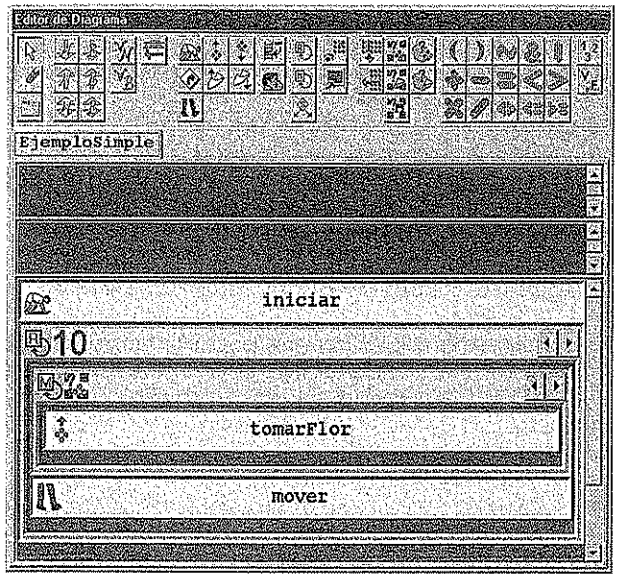


Figura 14.1

En cualquier lenguaje visual, la inserción de una componente es un proceso que se realiza en dos pasos: primero debe seleccionarse el componente deseado y luego debe indicarse el lugar donde se lo desea insertar.

En Visual Da Vinci, a medida que se incorporan los componentes, el lenguaje va construyendo en la ventana del Editor de Código, el programa correspondiente.

La figura 14.2 muestra el programa generado en forma visual.

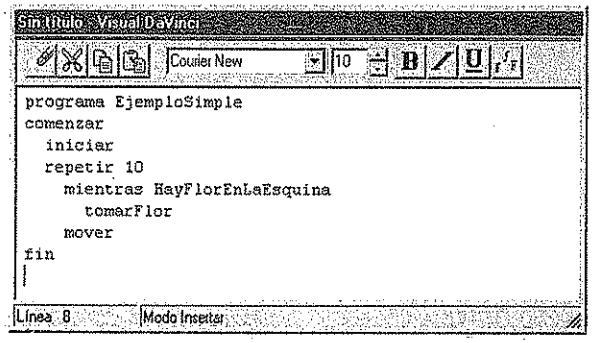


Figura 14.2

Sin embargo, Visual Da Vinci no es un lenguaje orientado a eventos, pues sus componentes no han sido pensados para reaccionar a ellos. El objetivo del Editor de Diagramas es el de proveer al programador de una herramienta visual que le facilite la codificación de los programas.

Los componentes de un lenguaje orientado a eventos están capacitados para reaccionar ante hechos externos y es tarea del programador definir el comportamiento adecuado según el problema a resolver.

Los lenguajes orientados a eventos son visuales como forma de proveer al programador de algún mecanismo para especificar las reacciones de los componentes frente a los eventos.

Varios de ellos, por ejemplo Visual Basic, Delphi y Visual C, utilizan una interfaz con el programador organizada como se indica en la Figura 14.3.

Paleta de Componentes	
Eventos y propiedades de cada Componente	Zona donde se construye la interfaz con el usuario

Figura 14.3

En la parte superior se encuentra la **paleta de componentes,** que contiene todos los objetos disponibles en el lenguaje. El programador puede personalizar esta paleta quitando los elementos que no desee utilizar y/o agregando nuevos componentes, ya sean definidos por él o importados de otros lenguajes.

A diferencia del Visual Da Vinci, las componentes que se pueden encontrar aquí son mucho más que una simple reproducción visual de la sintaxis del lenguaje. Se trata de elementos, como por ejemplo un botón o un menú desplegable, con un comportamiento definido para los cuales debe indicarse su reacción.

En el caso del botón, el efecto de "hundirse" al presionar con el mouse sobre él es un comportamiento definido que no necesita ser implementado. La tarea del programador es indicar cuál debería ser la reacción del botón al ser presionado. Para poder hacer esto, primero es necesario incorporarlo a la aplicación, pegándolo en la **zona de construcción de la interfaz con el usuario.**

A la izquierda suele presentarse una ventana, denominada **Inspector de Objetos**, que permite conocer los eventos a los que cada componente de la aplicación puede reaccionar, así como también sus propiedades. En este lugar es donde se le asocia el comportamiento definido

por el usuario al objeto. Así como los eventos de un objeto manejan sus reacciones, las **propiedades** del objeto manejan su apariencia. Algunos ejemplos de propiedades del botón son: color, tamaño, título que se visualiza en su interior, etc. El Inspector de Objetos permite modificarlas en tiempo de diseño, dándole al componente el aspecto deseado.

A medida que se inserten los objetos en la aplicación, el lenguaje de programación generará el código fuente correspondiente. Se puede notar que el código a desarrollar surge de la especialización de componentes predefinidos. Esto facilita el desarrollo del software ya que el usuario solo debe preocuparse por las "reacciones" deseadas.

Esto lleva a que buena parte del código se escriba en forma automática, lo que facilita la legibilidad y mantenimiento del software.

Paradójicamente, las reacciones que el programador debe especificar terminarán finalmente expresadas en un lenguaje imperativo.

Numerosos lenguajes de programación visuales permiten la programación orientada a eventos (Visual Basic, Delphi, Visual C++, JAVA).

A modo de introducción a lo que se expondrá en el capítulo siguiente, la Figura 14.4 muestra el esquema inicial de la pantalla del ambiente Delphi:

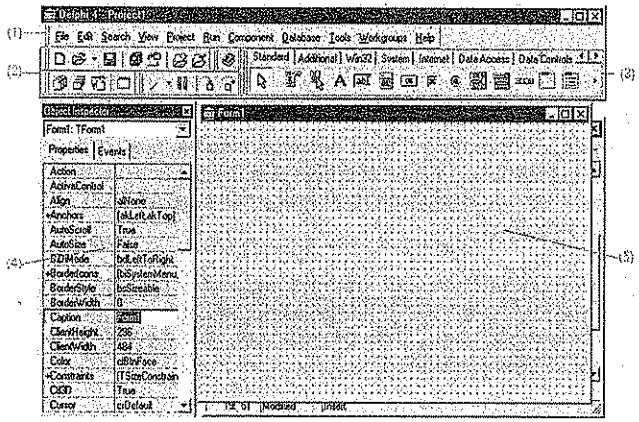


Figura 14.4

La organización de la interfaz es similar a la indicada en la figura 14.3.

En la Figura 14.4 se presentan tres sectores claramente separados:

En el sector superior se encuentra la **paleta de componentes** (3) junto con las opciones de manejo del ambiente: el menú principal (1) y los botones aceleradores (2) que permiten acceder de manera más rápida a algunas opciones del menú principal.

A la izquierda se halla la ventana correspondiente al **Inspector de Objetos** (4), que permite trabajar con todos los componentes de la aplicación, a través de la modificación de la apariencia y comportamiento de cada una de ellas.

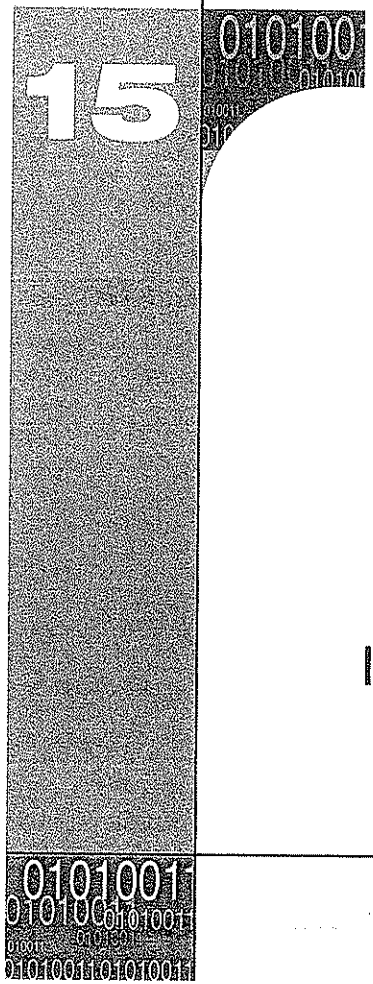
A la derecha se puede ver una ventana que lleva el título `Form1` (5). Esta ventana es conocida como **formulario** y constituye la pantalla donde se desarrollará la interfaz de la aplicación. Este formulario también es un componente y sobre él pueden insertarse o "pegarse" otros componentes obtenidos de la paleta (3), según lo requiera la aplicación.

Conclusiones

Se han expuesto sintéticamente las características destacables de los lenguajes visuales, ejemplificando brevemente con Visual Da Vinci.

Se ha presentado conceptualmente el paradigma de programación orientada a eventos, de modo de fundamentar la introducción al lenguaje Delphi que se hará en el capítulo 15.

Escapa al alcance de este texto profundizar en otros paradigmas de programación, que el lector seguramente utilizará en el desarrollo de su vida profesional.



Capítulo 15

Introducción a la programación en Delphi

15

Introducción a la programación en Delphi

Objetivo

En este capítulo se brindará una introducción a la programación en lenguaje Delphi con el objetivo de ejemplificar los conceptos explicados en el capítulo anterior.

Se ha seleccionado este lenguaje visual porque a través de su sintaxis Pascal permitirá comparar las soluciones implementadas mediante un lenguaje visual con lo realizado en la primera parte del libro.

El eje temático del capítulo es la presentación de los componentes básicos de Delphi, con el objetivo de capacitar al lector en el desarrollo de aplicaciones sencillas.

Los lectores interesados en profundizar el tema pueden consultar la siguiente dirección: http://lidi.info.unlp.edu.ar/~catedras/seminariob/capitulos.htm

15.1 Introducción

Como se explicó en el capítulo anterior, los **lenguajes visuales** extienden las posibilidades de los lenguajes convencionales incorporando elementos nuevos, que poseen un comportamiento predefinido, orientados a facilitar el diseño de la interfaz de la aplicación. Estos elementos, también llamados **componentes**, pueden modificarse tanto en la apariencia como en la respuesta esperada al interactuar con ellos.

Esto le permite al programador dedicar su atención al análisis y diseño de la solución, así como a la selección de las estructuras de datos adecuadas al problema, reduciendo de esta forma el tiempo de desarrollo.

El uso de herramientas que combinen lo visual con lo algorítmico llevó a cambiar el estilo de programación estructurada convencional hacia lo que se conoce como **programación orientada a eventos**.

Así como la programación estructurada se basa en la modularización, la programación orientada a eventos gira en torno al momento en que las señales del mundo real ocurren.
Ejemplos de eventos son: el click hecho sobre un botón, el ingreso de un valor por parte del usuario y la selección de una opción dentro de un menú desplegable.

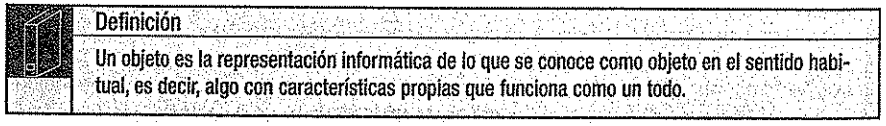
Dentro de este nuevo paradigma, el programador podrá incorporar un botón a su aplicación, darle la apariencia deseada e indicar, por ejemplo, qué hacer al clickear sobre él. Es importante destacar que el aspecto visual es manejado ahora por el lenguaje de programación. Esto último permite reducir el tiempo de desarrollo, ya que no solo resuelve el problema de la interfaz con el usuario, sino que facilita el mantenimiento del software y reduce la posibilidad de error.

Esta es la razón del éxito de los lenguajes visuales. Su facilidad de uso los convierte en una herramienta de desarrollo muy potente.

Para comprender mejor este paradigma se ha seleccionado el lenguaje Delphi por estar basado en Pascal y poseer los beneficios antes mencionados.

15.2 Nociones básicas

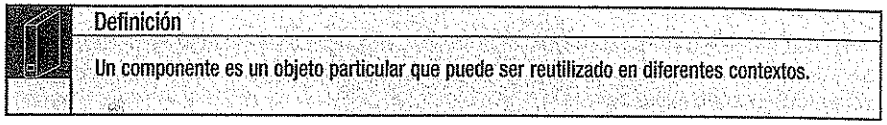
Antes de comenzar a programar en Delphi es necesario definir algunos conceptos básicos:



Definición

Un objeto es la representación informática de lo que se conoce como objeto en el sentido habitual, es decir, algo con características propias que funciona como un todo.

Ejemplos de objetos del mundo real pueden ser: una silla, un reloj o un termostato. En informática también se llama objeto a un botón, una ventana o un menú.

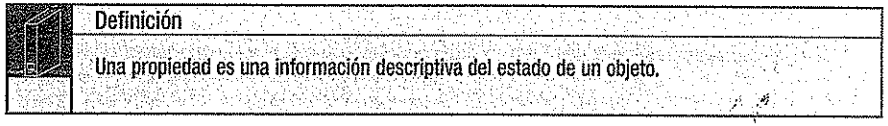


Definición

Un componente es un objeto particular que puede ser reutilizado en diferentes contextos.

Por lo general, estos conceptos son utilizados como sinónimos ya que la gran mayoría de los objetos que utiliza Delphi son diseñados y desarrollados a fin de ser reutilizados de diferentes maneras y en diferentes aplicaciones.

15.3 Propiedades y Eventos



Definición

Una propiedad es una información descriptiva del estado de un objeto.

Ejemplos de propiedades del objeto mesa podrían ser su altura, color, cantidad de patas, etc. En el caso del objeto formulario, algunas de sus propiedades podrían ser su título, alto y ancho.

Las propiedades brindan una descripción del objeto pudiendo ser consultadas y en ocasiones modificadas.

En el ejemplo del reloj, se puede considerar que posee una propiedad esencial: la hora actual. Este valor puede ser consultado o modificado en caso de ajustar el reloj a la hora oficial.

En el ejemplo del termostato, se puede decir que sus propiedades fundamentales son: la temperatura deseada y la temperatura real. La primera puede consultarse mirando el indicador correspondiente o bien puede modificarse por una nueva temperatura. Por el contrario, la temperatura real de una habitación no puede modificarse ya que depende de ciertas condiciones que son ajenas al termostato.

Las propiedades que pueden ser consultadas pero no pueden ser modificadas se conocen como de **solo lectura** o **lectura exclusiva**.

También sería posible encontrar un ejemplo de propiedad de **escritura exclusiva**, es decir, aquellas que pueden modificarse pero no consultarse, pero esto es menos corriente.

Desde el punto de vista de la informática, los objetos también poseen propiedades, de las cuales tal vez la más utilizada sea el título (`Caption`). En un botón, el título es el literal que aparece en él. En una ventana, se trata del título que aparece en la barra superior del contenido de la misma.

En Delphi es posible manipular directamente las propiedades en tiempo de desarrollo mediante el **Inspector de Objetos**. Para visualizar las propiedades de un objeto basta con seleccionarlo, clickeando sobre él.

Es importante destacar que el Inspector de Objetos permite visualizar la mayor parte de las propiedades de cada objeto, pero generalmente **no aparecen todas**.

También es posible agregar propiedades a los objetos ya definidos. Por ejemplo, si se incorpora una alarma al reloj del ejemplo, se tienen otras propiedades para el objeto reloj: la hora de la alarma y el hecho de que la alarma esté activa o no, situación que se puede consultar y también modificar. La incorporación de la alarma puede hacer que ciertas condiciones produzcan algunas reacciones. De esta forma, cuando se llega a la hora de la alarma y esta se encuentra activada, suena una señal.

Estas reacciones que se producen bajo ciertas condiciones son desencadenadas por eventos.

15.4 Eventos y métodos

Definición

Un evento es un hecho que se produce en un momento dado bajo ciertas condiciones y puede desencadenar reacciones.

En el ejemplo del termostato: cuando la temperatura ambiente sobrepasa la temperatura deseada en algunos grados (evento), la calefacción se desconecta (reacción); cuando la temperatura ambiente es inferior a la deseada en algunos grados (evento), la calefacción vuelve a entrar en funcionamiento (reacción).

Esto también ocurre cuando se trata de objetos informáticos.

En el caso de un botón, el evento más utilizado es el que ocurre al clickear con el mouse sobre él (`OnClick`). Este evento `OnClick` se halla presente en todas las componentes de Delphi.

Para visualizar y/o definir los eventos a los cuales puede reaccionar un componente, debe seleccionarse la lengüeta "Events" del Inspector de Objetos como se muestra en la Figura 15.1.

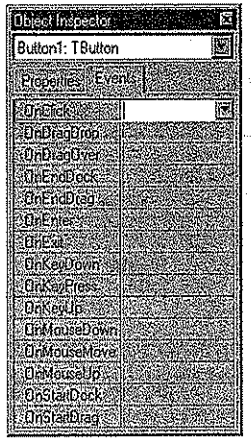


Figura 15.1

Para describir la reacción que produce un evento, se utiliza un método.

Un **método** es una descripción, mediante una secuencia de instrucciones, de lo que debe hacerse para obtener el resultado esperado.

Resumiendo lo dicho hasta ahora, si se desea poner el título "Mi primer ejemplo" al formulario principal, habitualmente llamado `Form1`, existen dos formas de hacerlo:

Directamente a partir del Inspector de Objetos: cambiando la propiedad `Caption`. Al hacerlo se verá cambiar el título inmediatamente.

Indirectamente: escribiendo un método que solicite el cambio de título mediante la instrucción:

```
Form1.Caption :=  "Mi primer ejemplo";
```

En este caso, el título se modificará cuando se llame al método.

15.5. Esquema de un programa Delphi

Antes de comenzar a trabajar en Delphi resulta de fundamental importancia discutir el formato de un programa en este lenguaje.

A diferencia de lo hecho hasta ahora en Pascal, el programa principal se ha transformado en un proyecto (*Project*) formado por una o más unidades (*Unit*).

Cada unidad tiene la misma forma que en Pascal. Esto es:

Código

```
Unit nombre_de_la_unidad;
interface
   { declaraciones públicas }
implementation
   { área privada. Lo aquí declarado sólo es conocido dentro de
     la unidad }
end.
```

En Delphi se realiza una asociación automática entre cada ventana o pantalla de salida y la unidad donde están declarados los componentes que contiene.

Por lo tanto, una aplicación nueva está formada principalmente por: una ventana con su unidad asociada que por defecto se llama `Unit1.pas` y un archivo con extensión `DPR` que contiene el programa principal y hace referencia a la unidad anterior, por defecto llamado `Project1.dpr`.

Siguiendo con esta idea, una aplicación que requiera varias ventanas de salida llevará a la incorporación de formularios (uno por cada ventana) que traerán asociadas sus respectivas unidades. Asimismo, es posible crear unidades que no tengan parte visual (no están asociadas a ningún formulario) e incorporarlas al proyecto.

Para poder acceder a lo declarado dentro de una unidad se utiliza la cláusula `Uses` seguida del nombre de la unidad que se desea referenciar.

15.6 El primer ejemplo

Se comenzará con un ejemplo muy simple al solo efecto de analizar las etapas necesarias para ejecutar un programa.

Ejemplo 1: Se desea visualizar una ventana titulada "Primer ejercicio" conteniendo un botón con el título "Haga clic aquí". Dicho botón, al ser presionado, debe cambiar su título a "Clic".

15.6.1 Creando un nuevo proyecto

Para comenzar una aplicación nueva, se debe seleccionar **File/New Application.**

Es aconsejable guardar la aplicación en un directorio previsto para ello. De esta forma, se evitará que el módulo ejecutable quede almacenado junto con el compilador Delphi. Para salvar la aplicación puede utilizar la opción **File/Save All.**

El lector comprobará que esta operación utiliza dos cuadros de diálogo. El primer cuadro de diálogo de grabación solicitará un nombre y un directorio para la unidad creada, por defecto llamada "`Unit1.pas`". Si bien puede conservar este nombre, se recomienda modificarlo por otro más representativo de la tarea a realizar, por ejemplo, "Primer_Ejemplo". Antes de presionar el botón **Save**, no olvide seleccionar el directorio adecuado.

El segundo cuadro de diálogo solicita un nombre y un directorio para el proyecto que representa a la aplicación. El nombre por defecto es "Project1.dpr". Así como en el paso anterior, se sugiere, además de seleccionar el mismo directorio que antes, asignarle un nombre más representativo como podría ser "Ejemplo1.dpr". Este es el nombre que llevará el módulo ejecutable. Para finalizar, seleccione nuevamente el botón **Save**.

A partir de este momento, ya se dispone de un formulario en blanco para poder comenzar a trabajar.

15.6.2 Ejecución del programa

Para ejecutar el programa puede escogerse entre: la opción Run/Run del menú principal, o la tecla F9 o el acelerador señalado en la Figura 15.2.

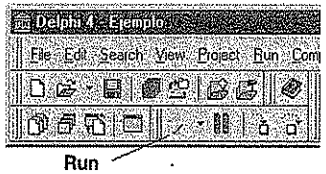


Figura 15.2

Si se ejecuta la aplicación se verá una pantalla en blanco que puede ser minimizada, agrandada, desplazada, etc.

Nótese que cuando la aplicación se está ejecutando, el Inspector de Objetos no aparece en pantalla y en el extremo superior izquierdo aparece al lado del nombre del proyecto el texto (Running) como lo muestra la Figura 15.3.

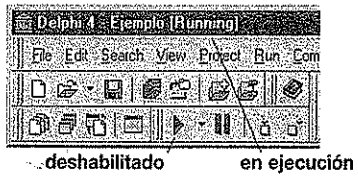


Figura 15.3

Dado que aún no se ha programado nada, resulta obvio que el lenguaje incorpora código que permite crear el proyecto.

Para poder continuar con este primer ejemplo, es necesario terminar con la ejecución actual. Para ello, debe utilizarse el botón del extremo superior derecho de la ventana con título `Form1` (marcado con "X"). Al terminar la ejecución, el entorno de desarrollo volverá a aparecer.

15.6.3 Incorporación de componentes

Para incorporar un componente al formulario basta con seleccionarlo en la paleta de componentes clickeando sobre él y luego ponerlo en la ventana, clickeando sobre ella.

Esto permitirá obtener un formulario como el de la Figura 15.4.

Si se ejecuta nuevamente la aplicación, puede verse que el botón se hunde al ser pulsado y luego vuelve a su posición elevada.

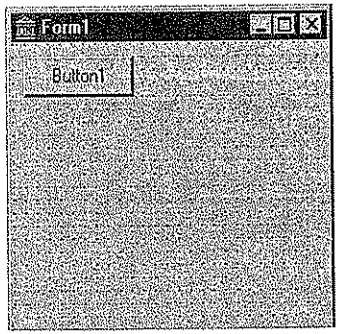


Figura 15.4

15.6.4 Modificación de propiedades

Para modificar el título del botón es necesario tenerlo seleccionado. Tal vez esta sea su condición, pero si no es así, para seleccionarlo sólo hay que clickear una vez sobre él.

La Figura 15.5 muestra que cuando un componente está seleccionado, sus bordes aparecen marcados por puntos de selección de color negro y en el Inspector de Objetos pueden verse sus propiedades y eventos.

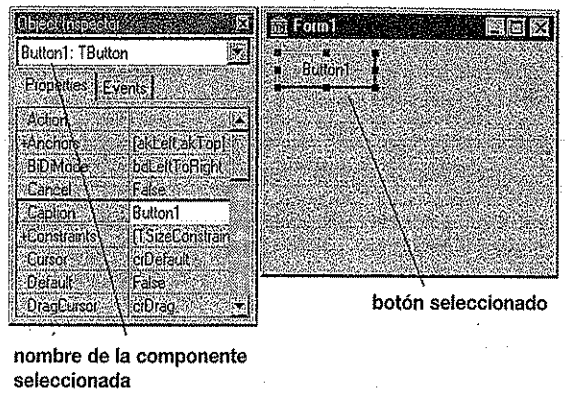


Figura 15.5

Para modificar el título del botón basta con cambiar el valor de su propiedad `Caption`. Donde dice `Button1` se debe escribir `Haga Click aquí`.

Para cambiar el título del formulario debe procederse de la misma forma. Se recuerda que para poder acceder a las propiedades del formulario es necesario que esté seleccionado. Antes de realizar cualquier modificación es importante verificar el nombre del componente que aparece en la parte superior del Inspector de Objetos.

El lector habrá constatado que la selección del formulario no provoca la visualización de los puntos de selección. En efecto, estos puntos de selección sirven principalmente para cambiar, de forma interactiva, el tamaño de los componentes. Sin embargo, los formularios son ventanas normales que pueden redimensionarse directamente.

Antes de ejecutar nuevamente se añadirá al botón el comportamiento esperado al ser presionado.

15.6.5 Agregando reacciones a eventos

Para agregar reacciones a eventos, es necesario seleccionar primero el componente con el que se desea trabajar. En el ejemplo, dado que el evento es el Clic sobre el botón, dicho botón es quien recibirá el evento y por lo tanto debe ser seleccionado.

Presionando sobre la lengüeta "Events" del Inspector de Objetos aparecerán los eventos más habituales del botón.

El primer evento es `OnClick`. Este evento se activa cuando se pulsa el botón y es aquí donde debe definirse qué hacer en este caso.

Para incorporar este nuevo comportamiento es necesario escribir un proceso asociado con este evento. La relación entre el proceso y el evento es tarea del Inspector de Objetos. Para saber dónde incorporar el nuevo código bastará con dar un doble click en el casillero que aparece a la derecha, como lo muestra la Figura 15.6.

Ante esta acción, Delphi visualizará automáticamente la pantalla de la Figura 15.7 correspondientes al editor de código.

El editor de código es un editor de texto adaptado para trabajar con el lenguaje Delphi (Pascal). Particularmente, resalta la sintaxis utilizando diferentes atributos; por ejemplo, negrita para las palabras claves y color azul para los comentarios.

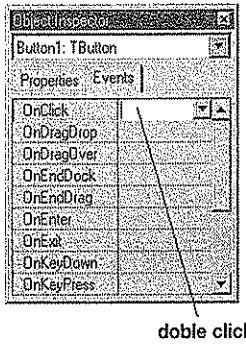


Figura 15.6

Si bien la figura 15.8 muestra el lugar donde se debe insertar el comportamiento deseado, el lector puede verificar que en la parte superior se hallan varias instrucciones generadas automáticamente. Al solo efecto de preservar la simplicidad de este primer ejemplo, dichas instrucciones no serán analizadas aquí.

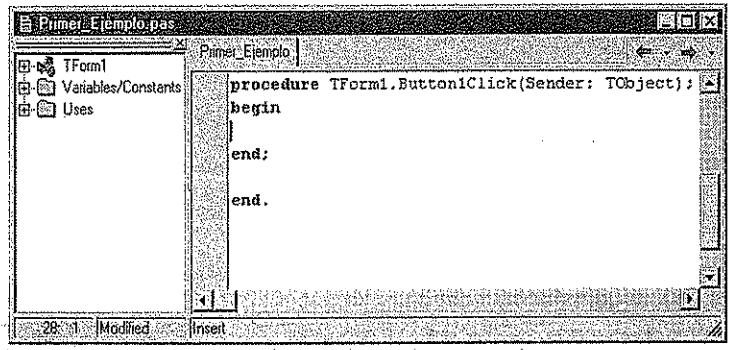


Figura 15.7

El procedimiento que resta definir solo contiene una instrucción y es la siguiente:

```
Button1.Caption := 'Click';
```

En esta instrucción hay varias cosas que analizar:

a) `Button1` es el nombre del botón. Este identificador es el valor indicado en su propiedad `Name` y puede verse desde el Inspector de Objetos.

b) Como se explicó anteriormente, `Caption` es la propiedad del botón que representa el título o texto que se visualiza dentro de él.

c) `Button1.Caption` muestra la sintaxis que debe respetarse dentro del código para acceder a una propiedad de un componente. Es importante no confundir esta notación con la utilizada para acceder a los campos de un registro. Aquí, `Button1` es el componente y `Button1.Caption` representa la propiedad donde está guardado el título que se visualiza dentro de él.

d) La asignación no se reflejará hasta que la instrucción no se haya ejecutado.

Por lo tanto, `Button1.Caption := 'Click'` permitirá modificar el título de la manera esperada. Nótese que el valor asignado es un *string* y su sintaxis es la utilizada por Pascal, es decir, encerrado entre comillas simples.

Verifíquese ahora el funcionamiento de la aplicación utilizando la tecla F9.

Ejercicio 1:

Se sugiere al lector que resuelva el siguiente ejercicio para reforzar los conocimientos aprendidos hasta el momento. La solución de este ejercicio se encontrará en la página web del libro.

Desarrollar un programa en Delphi que contenga una ventana con el título "Ejercicio 1". Dentro de ella se encuentran dos botones titulados: "Opción 1" y "Opción 2", como lo muestra la Figura 15.8.

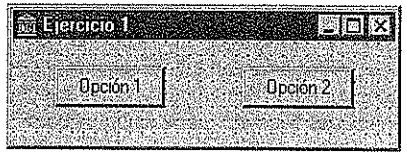


Figura 15.8

Al presionar el botón de la izquierda, el título de la ventana debe cambiar a "Opción 1 seleccionada" y al presionar el de la derecha a "Opción 2 seleccionada".

15.7 Aspectos generales de las componentes de Delphi

Si bien cada componente de Delphi posee sus características propias, existen propiedades y eventos que habitualmente se repiten. Dado que es difícil recordar los atributos de cada componente en particular, especialmente en la etapa de iniciación a la programación visual, resulta de interés efectuar un análisis general de las componentes.

15.7.1 Propiedades más comunes

Propiedad Name

Toda componente posee la propiedad `Name`. Su valor es equivalente al nombre de una variable de Pascal y es la manera de acceder al componente desde el código.

Cuando una componente es insertada dentro del formulario, Delphi se encarga de darle un nombre que está formado por la concatenación de un identificador y un número correlativo que permite distinguirla de sus semejantes. Por ejemplo, si se pegan tres botones sobre un formulario en blanco, el primero recibirá por defecto el nombre `Button1`, el segundo `Button2` y el tercero `Button3`. Si el programador lo desea, podrá cambiar esta identificación a través del Inspector de Objetos.

Propiedad Caption o Text

La propiedad `Caption` o la propiedad `Text` permite visualizar un *string* dentro del componente.

Estas propiedades son excluyentes entre sí. Es decir, que las componentes que poseen la propiedad `Caption` no poseen la propiedad `Text`.

La diferencia radica en la atribución del usuario para modificar su valor durante la ejecución de la aplicación.

La propiedad `Caption` se refiere a un **texto fijo** que el usuario no puede modificar en tiempo de ejecución como, por ejemplo, el título del formulario o el título del botón. Por el contrario, la propiedad `Text` se refiere a un *string* que el usuario puede visualizar y modificar en ejecución. Esto se describe en la sección siguiente con mayor detalle.

Ya sea que la componente posea `Caption` o `Text`, su valor por defecto coincide con el de la propiedad `Name`. Es habitual que un programador Pascal que se inicia en la programación Delphi confunda estas propiedades llegando, por ejemplo, a borrar el nombre de una componente al intentar borrar el texto que en ella se visualiza. Esta acción provocará un error, ya que la aplicación no tiene forma de referenciar una componente sin nombre.

Propiedades Visible y Enabled

La propiedad `Visible` contiene un valor booleano que indica si el componente debe visualizarse dentro del formulario cuando la aplicación se ejecuta. Nótese que su efecto solo se aprecia en ejecución.

La propiedad `Enabled`, en lugar de ocultar al componente, permite deshabilitarlo. Por ejemplo, si se trata de un botón, al presionarlo no ocurrirá ningún evento; si se trata de un casillero donde el usuario habitualmente ingresa información, no podrá hacerlo.

Propiedades que manejan la apariencia del componente

Las propiedades `Width` y `Height` representan el ancho y el alto del componente respectivamente. Su valor está expresado en píxeles.

También pueden hallarse propiedades como `Color`, que permite indicar un color para el componente; `Font` que permite modificar la fuente del texto, etc.

15.7.2. Eventos más comunes

`OnClick`: se activa al clickear con el mouse sobre la componente

`OnMouseDown`: se activa cuando se presiona uno de los botones del mouse sobre la componente. A diferencia del `OnClick` permite reconocer cuál de los botones del mouse fue presionado.

`OnMouseUp`: se activa cuando se deja de presionar uno de los botones del mouse sobre la componente. A diferencia del `OnClick` permite reconocer cuál de los botones del mouse fue presionado.

`OnMouseMove`: se activa cuando el cursor del mouse pasa sobre la componente.

`OnEnter`: se activa cuando la componente recibe el foco. En una aplicación que se está ejecutando solo una componente por formulario puede estar activa a la vez o tener el foco.

`OnExit`: se activa cuando la componente pierde el foco.

`OnKeyPress`: se activa cuando la componente tiene el foco y se presiona una tecla cualquiera.

15.8 Entrada/salida en Delphi

Pascal posee una sintaxis muy simple, a través de las instrucciones `Read` y `Write`, que le permite manejar la entrada/salida de información.

Sin embargo, en Delphi estas acciones deben ir acompañadas por la interfaz correspondiente. Por este motivo, existen diferentes alternativas para ingresar o visualizar datos, según los requerimientos del problema a resolver.

Por una cuestión de uniformidad, todas las componentes manejan la información en formato *string*. Esto lleva a utilizar funciones de conversión cuando se trata de valores numéricos.

A continuación se describen diferentes alternativas básicas que permiten realizar entrada/salida en Delphi.

15.8.1 Componentes básicos

`Label` (Paleta Standard)

Permite visualizar una única línea de texto. Si bien habitualmente suele ser utilizada como título para alguna otra componente, también puede resultar de utilidad al desear mostrar algún resultado. El *string* que se visualiza es el valor de la propiedad `Caption`.

`Edit` (Paleta Standard)

Permite ingresar por teclado y/o visualizar en su interior, una única línea de texto.
Su uso habitual es recuperar la información ingresada por el usuario. También puede utilizarse solo como salida seteando su propiedad `ReadOnly` en verdadero.
La información a mostrar y/o recuperar se encuentra almacenada en la propiedad `Text` y posee formato texto (*string*).

`MaskEdit` (Paleta Additional)

Es similar a la componente `Edit`, pero incorpora una propiedad muy importante: una máscara de validación. De esta forma, la componente realiza automáticamente la validación correspondiente, impidiendo el ingreso de caracteres no deseados. Dicha validación se realiza caracter por caracter.

La máscara consiste de tres campos separados por punto y coma. La primera parte de la máscara es la máscara en si misma. La segunda parte es un caracter que determina si los caracteres literales de la máscara deben salvarse con los datos o no. La tercera parte de la máscara es el caracter utilizado para representar los lugares donde el usuario puede ingresar la información.

15.8.2 Procesos predefinidos que permiten visualizar un string

`ShowMessage:`

Es un procedimiento que posee la siguiente sintaxis

```
procedure ShowMessage(const Msg: string);
```

Su uso es similar al de la instrucción `Write` de Pascal ya que permite ver en pantalla el valor del *string* que lleva como parámetro, pero con la diferencia de que para hacerlo utiliza una ventana propia.

El parámetro `Msg` es el *string* que aparece en ella. El nombre del archivo ejecutable de la aplicación aparecerá como el título de la ventana.

`MessageDlg:`

Es una función con la siguiente sintaxis:

```
function MessageDlg(const Msg: string; AType: TMsgDlgType;
  AButtons: TMsgDlgButtons;
  HelpCtx: Longint): Word;
```

que permite visualizar una ventana con el valor del parámetro `Msg`.

El parámetro `AType` determina el tipo de ventana a utilizar. Sus valores posibles son: `mtWarning`, `mtError`, `mtInformation`, `mtConfirmation` y `mtCustom`. El valor elegido permitirá modificar la pariencia de la ventana (color, símbolos gráficos que contiene, etc.).

El parámetro `AButtons` determina cuáles son los botones a visualizar. Su valor es un conjunto de valores de tipo enumerativo, de manera que se pueda incluir una cantidad variable de botones en la ventana.

Los valores que pueden incluirse en el conjunto son: `mbYes`, `mbNo`, `mbOK`, `mbCancel`, `mbHelp`, `mbAbort`, `mbRetry`, `mbIgnore`, `mbAll`.

Además de los valores individuales, existen tres conjuntos de botones predefinidos: `mbYesNoCancel`, `mbOkCancel`, `MbAbortRetryIgnore`.

El parámetro `HelpCtx` determina la página de Help disponible para esta ventana de ayuda.

La función `MessageDlg` retorna el valor del botón seleccionado por el usuario. Sus valores posibles son: `mrNone`, `mrAbort`, `mrYes`, `mrOk`, `mrRetry`, `mrNo`, `mrCancel`, `mrIgnore` y `mrAll`.

Ejemplo 15.1

```
if MessageDlg('¿Desea salir?', mtConfirmation,
              [mbYes, mbNo], 0) = mrYes then        (1)
begin
   { escriba aquí el proceso de finalización }
   MessageDlg('Terminado', mtInformation, [mbOk], 0); (2)
end;
```

La línea (1) muestra una ventana de confirmación con dos botones. Si el usuario presiona el botón "Yes", el proceso de finalización será ejecutado mostrando una ventana de información con un único botón de "OK".

Es importante rescatar del ejemplo anterior que la manera en que Delphi permite invocar a una función. Las funciones, al igual que los procedimientos, pueden ser utilizadas como sentencias completas; de ser así, el valor retornado simplemente es descartado.

15.9 Primer ejemplo utilizando entrada/salida

A continuación se define un ejemplo muy sencillo que utiliza las componentes descritas previamente con el objetivo de clarificar su uso.

Ejemplo 2: Desarrollar un programa en Delphi que calcule la suma de dos valores enteros que se ingresan por teclado.

Se comenzará con un proyecto nuevo mediante la opción **File\New Project** del menú principal. De ser necesario, se deberá salvar la aplicación anterior, a fin de no perder el trabajo hasta aquí realizado.

Es importante salvar este proyecto nuevo, dándole nombre a cada módulo como ya se describió previamente. Es un problema habitual al comenzar a utilizar Delphi que el programador confunda el directorio de trabajo; se solicita especial atención en este tema.

Una vez obtenida la ventana principal se deberán pegar las componentes que se indican en la Figura 15.9. En dicha Figura se enuncian las propiedades que deben modificarse.

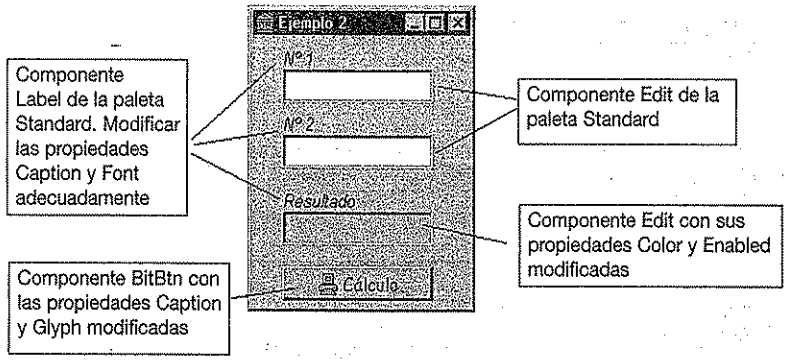


Figura 15.9

El funcionamiento de la aplicación será el siguiente: el usuario deberá ingresar los dos números enteros, uno en cada uno de los casilleros blancos (`Edit1` y `Edit2`). Luego, presionando el botón, la suma se reflejará en el casillero gris (`Edit3`).

Con respecto al ingreso de información no es necesario programar nada ya que la componente `Edit` lo realizará automáticamente. Por lo tanto, solo resta indicar cómo hacer la suma. Esta tarea deberá ser indicada en el evento `OnClick` del botón como se detalla a continuación:


```
procedure TForm1.Button1Click(Sender: TObject);
Var Nro1, Nro2 : Integer;
begin
    Nro1 := StrToInt(Edit1.Text);
    Nro2 := StrToInt(Edit2.Text);
    Edit3.text := IntToStr(Nro1 + Nro2 );
end;
```

Como se dijo anteriormente, la información ingresada a través de `Edit` es de tipo *string* y por lo tanto debe ser convertida.

Se utilizaron las siguientes funciones:

`StrToInt`: recibe un *string* y lo devuelve convertido en un número entero. Si el *string* recibido como parámetro no contiene un número entero, se producirá un error de conversión (`EConvertError`).

`IntToStr`: recibe un entero y lo retorna convertido en un *string*.

Si bien no lo requiere la resolución del ejercicio 2, también se encuentran disponibles las funciones `FloatToStr` y `StrToFloat` que permiten trabajar con números que no son enteros.

Ya es posible ejecutar la aplicación y verificar su funcionamiento.

Ejercicio 2: Escriba un programa que, dada una secuencia de caracteres terminada en punto, calcule:

a) Porcentaje de caracteres que aparecen en la frase y que no son letras.

b) Cantidad de consonantes de la frase.

c) Longitud de la palabra más larga.

d) Se sugiere utilizar la interfaz de la Figura 15.10.

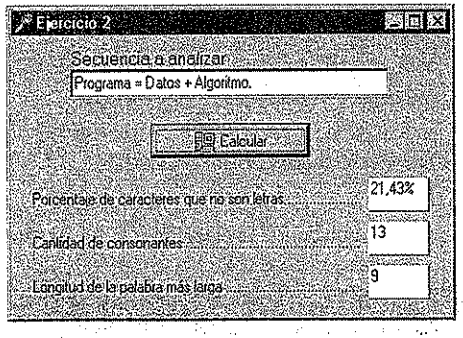


Figura 15.10

15.10 Manejo de excepciones

Como es sabido, los errores que se producen durante la ejecución de un programa llevan a que el mismo aborte. En Pascal, para controlarlos se incorpora código *ad hoc* a fin de prevenir que ocurra.

Esto trae aparejados dos grandes problemas: en primer lugar, oscurece el código desarrollado, ya que se mezclan las instrucciones dedicadas a resolver el problema con las requeridas para evitar los errores; en segundo lugar, disminuye la eficiencia del código pues las validaciones son realizadas independientemente de si se trata de una situación de error o no.

Por ejemplo, para trabajar con un archivo el programador debería tener en cuenta: la existencia del archivo, la validez de la ruta de acceso indicada, la integridad del archivo; además, si el archivo está en una red tendría que verificar que la conexión esté activa. Aunque el código para utilizar el archivo sea pequeño todas las verificaciones retardarían la ejecución del programa.

El manejo de las excepciones permite separar el código necesario para resolver el problema del código desarrollado para manejar los errores. De esta forma no se retrasa la ejecución del programa porque el código normal se ejecuta siempre, pero el código de manejo de errores se ejecuta solo cuando es necesario.

Definición

Una excepción es un evento que ocurre por un error del programa en tiempo de ejecución y es generada para indicarle al programador que han ocurrido errores que impiden la normal ejecución del programa.

Definición

El proceso que permite resolver los problemas generados por una excepción se denomina el manejador de dicha excepción.

Por ejemplo, en Pascal, para realizar una división entre dos números es común utilizar el siguiente código:

```
if  Z <> 0 then X := Y / Z
           else ResolverDivisionPorCero;
```

La verificación de que Z no sea cero antes de dividir no tiene que ver con el cálculo de la división, sino con evitar que el programa aborte al ejecutar una división por cero.

Si bien el ejemplo anterior es muy simple y solo requiere verificar el valor de Z, en un problema real puede tratarse de un problema complejo lo que llevaría a oscurecer y a reducir la eficiencia del código desarrollado.

Si bien el código del ejemplo anterior es correcto, Delphi provee la sentencia `try - except` como una alternativa más eficiente para resolver este problema.

Esta sentencia tiene dos partes

```
Try
     {bloque de instrucciones que resuelven el problema
     sin validaciones}
  except
      {manejo de los errores que pudieron producirse en
       el bloque anterior}
end;
```

a) Entre `try` y `except` se colocan las sentencias que se desean proteger. En esta parte, el programador podrá trabajar como si los errores de compilación no existieran.

b) Entre `except` y `end` se codifican los manejadores de excepciones que se consideren convenientes para resolver los problemas que puedan aparecer en el bloque anterior. Es importante destacar que este sector no se ejecutará a menos que un error se produzca. Es decir, en una situación normal solo se ejecutará el bloque de instrucciones entre `try` y `except`.

Volviendo al ejemplo anterior, el siguiente código muestra cómo resolverlo utilizando esta nueva instrucción:

```
try
  X := Y/Z;
except
  on EZeroDivide do ResolverDivisionPorCero;
end;
```

La sentencia anterior intenta dividir Y por Z y llama al procedimiento `ResolverDivisionPorCero` en caso de que ocurra una excepción provocada por una división por cero.

A continuación se detalla la sintaxis correspondiente a la sentencia `try-except`:

```
a)  Try
       { Bloque de instrucciones que se desea proteger }
    except
        on ( Tipo de excepción 1 ) do Manejador_Excepcion_1;
        on ( Tipo de excepción 2 ) do Manejador_Excepcion_2;
        ...
         else Manejador_para_las_demas;
    end;
```

Entre `try` y `except` se codifican las instrucciones que resuelven el problema y entre `except` y `end` se colocan los distintos manejadores de excepciones.

Si la ejecución de las sentencias que se encuentran entre `try` y `except` no presentan error alguno, ninguno de los manejadores que aparecen después del `except` será ejecutado, pasando el control a la primera línea fuera de la sentencia `try-except` (después de `end`).

Por el contrario, si algún error ocurre durante la ejecución de las instrucciones ubicadas entre `try` y `except`, el control pasará al bloque donde están ubicados los manejadores de excepciones. Si se ha definido un manejador para el tipo de error que ha ocurrido, dicho manejador tomará el control. Si tal manejador no ha sido definido, el control pasa a la sentencia `else`, si es que ha sido definida. Si tampoco se ha definido la sentencia `else`, el programa abortará.

Existe la posibilidad de anidar las sentencias `try-except` en cuyo caso, si una sentencia `try-except` requiere de un manejador no definido, Delphi lo buscará en la sentencia `try-except` contenedora. Este proceso se repite hasta hallar el manejador adecuado o hasta que no haya más sentencias `try-except` donde buscar, en cuyo caso el programa termina.

b)
```
Try
   { secuencia de instrucciones 1 }
 except
   { secuencia de instrucciones 2 }
end;
```

Al igual que en a) la secuencia de instrucciones 2 solo se ejecutará en caso de producirse un error. Nótese que en b) solo aparece un único manejador de excepción que será activado por **cualquier** error que se produzca.

Tipos de excepciones predefinidos

A continuación se describen algunos de los tipos de excepciones predefinidos en Delphi:

Tipo de excepción	Cuando ocurre
`EaccessViolation`	El procesador ha detectado un acceso inválido a una región de memoria. Puede ocurrir cuando se pretende acceder a una dirección de memoria utilizando un puntero con valor `nil` o cuando se accede a una región de memoria no reservada.
`EConvertError`	Alguna de las siguientes funciones: `StrToInt` o `StrToFloat` ha sido incapaz de convertir el *string* especificado en un entero válido o en un valor de punto flotante respectivamente.
`EDivByZero`	La aplicación ha intentado dividir un entero por cero utilizando el operador **div**.
`EInOutError`	El sistema operativo ha detectado un error de entrada/salida.
`EIntOverFlow`	Se ha producido un error de *overflow* en una operación entera.

`EInvalidCast`	Error al intentar convertir un objeto de n tipo en otro utilizando el operador **'as'**.
`EInvalidOp`	Es una excepción matemática de punto flotante. Aparece siempre que el procesador encuentre un resultado inexacto, un operador inválido o se haya producido *stack overflow*.
`EInvalidPointer`	La aplicación ha realizado una operación con un puntero inválido. Ocurrirá, por ejemplo, si la aplicación intenta liberar el mismo puntero dos veces.
`EOutOfMemory`	Este error ocurre cuando la aplicación desea reservar memoria en forma dinámica pero la memoria libre existente en el sistema resulta insuficiente para completar la operación requerida.
`EOverflow`	El resultado calculado es demasiado grande para ser almacenado en el lugar indicado y por lo tanto, la información se pierde.
`ERangeError`	Es una excepción matemática entera. Puede ocurrir o bien cuando el resultado de una expresión entera excede los límites del tipo entero esperado o cuando se intenta acceder a un elemento de un arreglo utilizando un valor de índice fuera del rango permitido.
`EZeroDivide`	Ocurre cuando se produce una división por cero.

Tabla 15.1

Retomando el Ejemplo 2, el ingreso de información es realizado a través de componentes de tipo `Edit`, es decir, de tipo texto sin validación del ingreso de caracteres. En caso de ingresar una palabra, en lugar de alguno de los números enteros esperados, se producirá un error al presionar el botón. Nótese que el componente `Edit` no distingue entre números y letras, por lo que el ingreso de datos no producirá error alguno. El problema se presentará al intentar convertir la palabra ingresada en un número utilizando la función `StrToInt`. El lector puede ejecutar nuevamente el programa verificando que no siempre funciona.

Para validar la información ingresada es posible escribir el código Pascal correspondiente de manera de evitar que el programa aborte al momento de realizar la conversión. Sin embargo, Delphi posee un mecanismo más eficiente para realizar esta tarea: las excepciones.

A continuación se modificará el procedimiento asociado al `OnClick` del botón utilizando el manejo de excepciones en Delphi como forma de resolver este problema.

Código

```
procedure TForm1.Button1Click(Sender: TObject);
Var Nro1, Nro2 : Integer;
begin
  Try
    Nro1 := StrToInt(Edit1.Text);
    Nro2 := StrToInt(Edit2.Text);
    Edit3.text := IntToStr(Nro1 + Nro2 );
  except
    on EConvertError do
        begin
            Edit3.text := '';
            MessageDlg('Operandos Inválidos',mterror,[mbabort],0);
        end;
  end;
end;
```

15.11 Analizando la unidad que maneja el formulario

En la sección 15.5 se describió la forma de una unidad en Pascal y se discutió la posibilidad de disponer de unidades que tuviesen o no un formulario asociado.

En esta sección se analizarán las distintas partes de una unidad que maneja un formulario. Suponiendo que se trata de una aplicación con una única ventana llamada `Form1`, la unidad asociada se compone de lo siguiente:

Código

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;

implementation
{$R *.DFM}
end.
```

Definición de la clase a la que pertenece el formulario

Al igual que lo descrito en la sección 15.5, se encuentra separada la parte de interfaz de la parte de implementación. Dado que aún no se ha desarrollado código alguno, esta última no contiene ningún módulo.

Sin embargo, el lenguaje ha comenzado la definición del formulario en la sección de interfaz. Dentro de ella se distinguen dos partes: una parte privada que comienza a continuación de la palabra `private` y otra pública, indicada por la palabra `public`.

En la sección privada pueden declararse variables y procesos que solo podrán ser accedidos por este formulario. No son visibles desde el exterior. Esta información corresponde a la representación interna y al comportamiento del formulario.

Por el contrario, lo declarado en la sección pública es conocido por aquellos que invocan al formulario.

El ambiente se reserva una parte dentro de esta declaración y es la que se encuentra delante de la palabra `private`. En esta región, el programador no puede incorporar código, ya que es considerada una zona exclusiva para Delphi.

Tanto en la zona pública como en la privada, lo correspondiente a la representación debe ser colocado antes que el comportamiento; es decir, las variables deben escribirse antes que los procedimientos o funciones.

En la zona de **interfaz** solo se colocan los encabezados de los módulos, mientras que en la parte de implementación se codifica el módulo completo.

Ejemplo 3 : Implementar un programa Delphi que permita visualizar los nombres de los días de la semana según la siguiente interfaz (Figura 15.11)

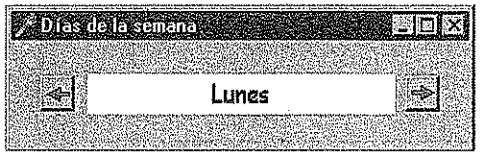


Figura 15.11

El botón que aparece a la derecha permite ver el nombre del día siguiente y el de la izquierda el del día anterior. Ambos botones realizan su tarea en forma circular.

Para resolver este problema es necesario registrar en algún lugar el día actual de manera que ambos botones conozcan su valor al momento de ser presionados.

Como este valor no necesita ser conocido fuera del formulario, es lógico declararlo en la región `private`. Además, se utilizará un procedimiento `VerNombre` que visualizará el nombre del día indicado.

La declaración de la clase a la que pertenece el formulario será la siguiente:

Código

```
TForm1 = class(TForm)
      SpeedButton1: TSpeedButton;
      SpeedButton2: TSpeedButton;
      Label1: TLabel;
    private
      { Private declarations }
      NroDia : Integer;
      procedure VerNombre;
      public
      { Public declarations }
  end;
```

Agregado por Delphi en forma automática al incorporar los componentes.

Ahora, es preciso indicar cómo modificar el valor de `NroDia`.

A continuación se detallan los eventos `OnClick` de cada uno de los botones:

Código

```
procedure TForm1.SpeedButton1Click(Sender: TObject);
{ Botón izquierdo }
begin
  NroDia := NroDia - 1;
  VerNombre;
end;
procedure TForm1.SpeedButton2Click(Sender: TObject);
{ Botón derecho }
begin
  NroDia := NroDia + 1;
  VerNombre;
end;
```

Para inicializar el valor de `NroDia` se utilizará el evento `OnActivate` del formulario. Este evento, característico del formulario, ocurre cuando el mismo se hace visible. En el ejemplo, esto ocurre cuando comienza la ejecución.

Código

```
procedure TForm1.FormActivate(Sender: TObject);
begin
   NroDia := 1;
   VerNombre;
end;
```


El procedimiento `VerNombre` no será incorporado por el Inspector de Objetos como los módulos anteriores, sino que deberá ser escrito por el programador.

Su definición es la siguiente, y debe ser escrita en la sección `implementation`.

Código

```
procedure TForm1.VerNombre;
const  Nombres : array[1..7] of string =
                 ( 'Lunes', 'Martes', 'Miércoles', 'Jueves',
                   'Viernes', 'Sábado', 'Domingo');
begin
   {$R+}
   try
       Label1.caption := Nombres[ NroDia ];
   except
       if NroDia > 7
         then  NroDia := 1
         else  if NroDia < 1 then NroDia := 7;
       Label1.caption := Nombres[ NroDia ];
   end;
end;
```

En la primera línea puede observarse que el nombre del procedimiento va precedido por el nombre de la clase a la que pertenece el formulario. Esto representa que el procedimiento forma parte del comportamiento del formulario y por lo tanto no necesita recibir la información del formulario como parámetro ya que siendo parte de él tiene acceso a toda su representación, ya sean componentes o variables internas, como `NroDia`.

Nótese, además, que el manejo de excepciones permite al programa dedicarse a mostrar el nombre del día de la semana y sólo chequea el valor de `NroDia` en caso de producirse un acceso inválido.

15.12 Compartiendo eventos. El uso del parámetro Sender

Todos los procedimientos cuya interfaz se genera en forma automática se caracterizan por tener un parámetro común: `Sender`.

Este parámetro permite saber cuál fue el componente que recibió el evento. En el ejemplo anterior, el evento que se produjo fue el click sobre el botón y por ello el parámetro `Sender` representa al botón involucrado.

A fin de poder ser aplicado a cualquier clase de componente, `Sender` está especificado como un objeto genérico; esto queda indicado al decir que pertenece a la clase `TObject`.

`TObject` es la clase más genérica a la que pertenecen todos los componentes de Delphi.

Dicho en otras palabras, el parámetro `Sender` expresado como un objeto de la clase `TObject`, no posee las propiedades de ningúna componente en especial. Para ello debe ser convertido adecuadamente utilizando la siguiente sintaxis:

```
Nombre_Clase( Sender );
```

Donde `Nombre_Clase` es el nombre de la clase en la que se quiere convertir al parámetro `Sender`.

Por ejemplo, si `Sender` hace referencia a una componente `Edit` y se desea conocer el texto que contiene, es necesario pasar el objeto `TObject` a un objeto perteneciente a la clase `TEdit` para poder acceder a su propiedad `text`.

La sintaxis a utilizar en la siguiente: `TEdit(Sender).Text`.

Ejemplo 4: Desarrolle una aplicación en Delphi con la interfaz de la figura 15.12. Al presionar sobre cualquiera de los botones, se debe visualizar un mensaje indicando el título (`Caption`) del botón presionado.

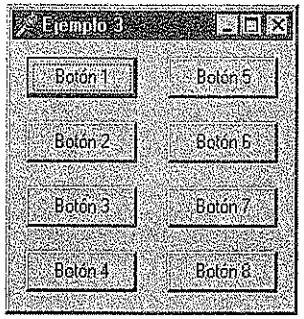


Figura 15.12

Una forma de resolver este problema sería escribir ocho procesos diferentes, uno para cada evento `OnClick` de cada botón. Cada proceso estaría formado por una única instrucción de la forma:

```
ShowMessage(  Nombre_del_boton.Caption  );
```

Otra forma más eficiente de resolver esto sería escribir un **único** proceso asociado a **todos** los eventos `OnClik`. Obviamente, el mensaje no puede ser fijo, pues todos los botones mostrarían el mismo resultado.

A continuación se muestra el código correspondiente al `OnClick` del primer botón.

Código

```
procedure TForm1.Button1Click(Sender: TObject);
Var EsteBoton : TButton;
begin
   EsteBoton := TButton( Sender );
   ShowMessage( EsteBoton.Caption );
end;
```

Los botones restantes no deben definir un proceso nuevo, sino que deben utilizar el Inspector de Objetos para referenciar al proceso existente. La Figura 15.13 muestra cómo hacer el enlace para el botón `Button2`. En lugar de clickear sobre el casillero en blanco, se selecciona el evento de la lista de eventos existentes Esto debe repetirse para los demás botones.

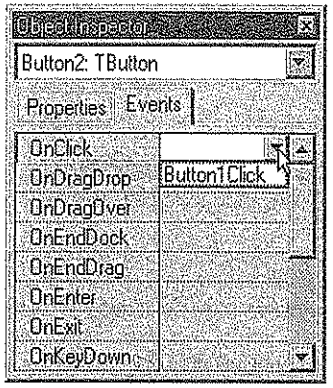


Figura 15.13

Ejercicio 4: Implemente una calculadora sencilla según la interfaz de la Figura 15.14. Por simplicidad, solo se realizan las operaciones elementales: suma, resta, multiplicación y división. La única forma de ingresar información es a través de los botones que aparecen en pantalla, es decir, no es posible incorporar caracteres por teclado.

La evaluación de la expresión se realiza al presionar el botón "=".

De ser posible, considere en su implementación la precedencia de los operadores de multiplicación y división con respecto a la suma y a la resta.

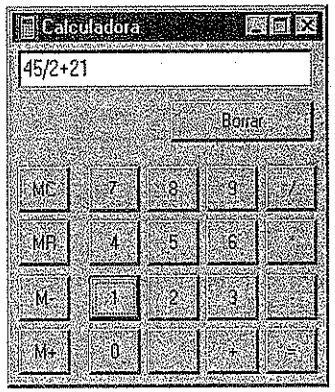


Figura 15.14

Conclusiones

Se ha brindado sintéticamente una introducción a la programación en lenguaje Delphi.

Se han presentado las componentes básicas de dicho lenguaje visual a fin de capacitar al lector en el desarrollo de aplicaciones sencillas. Los conceptos explicados en este capítulo son suficientes para rehacer los ejercicios implementados en Pascal en capítulos anteriores.

El lector podrá aplicar estos conocimientos sobre cualquier versión de Delphi de la cual disponga.

Conclusiones

Informática, tecnología y programación: una mirada al futuro.

La Informática es una disciplina (y una fuente laboral) que crece. Pese al enorme esfuerzo en la formación de recursos humanos en todo el mundo, aún hoy, la demanda y el costo laboral de los especialistas en Informática aumenta.

Es claro que el eje de este crecimiento está en la aplicación cada vez más extendida de la tecnología informática. Un punto de referencia cualquiera (desde los artefactos del hogar hasta los vuelos espaciales; desde los laboratorios automatizados hasta los efectos especiales de una película; desde un robot industrial o un avión de combate hasta un videojuego o un teléfono celular) tiene componentes informáticos. Más aún: el componente distintivo del comportamiento del aparato o instrumento es su software, es decir su programación.

Esto obliga a una gran integración de conocimientos propios de Informática con datos específicos del mundo real de las aplicaciones, para obtener productos informáticos útiles (y rentables).

La tecnología es entonces el motor del crecimiento de la Ciencia Informática.

Hoy en el mundo el área económica más importante es la que vincula la electrónica con las comunicaciones y con el software. Por ende esta integración y el conocimiento tecnológico que ella implica son el eje de la formación de recursos humanos y una de las fuentes laborales más importantes.

Dentro del desarrollo de sistemas de software la programación sigue estando en un núcleo del conocimiento básico. Paradójicamente, aún cuando se automatizara totalmente el proceso de codificación de módulos de sistemas (tal como es la tendencia en la producción industrial de software), seguirá siendo muy importante formar individuos capaces de pensar un problema real, abstraer sus características, modelar una solución posible sobre computadora, descomponerlo y especificarlo en forma algorítmica.

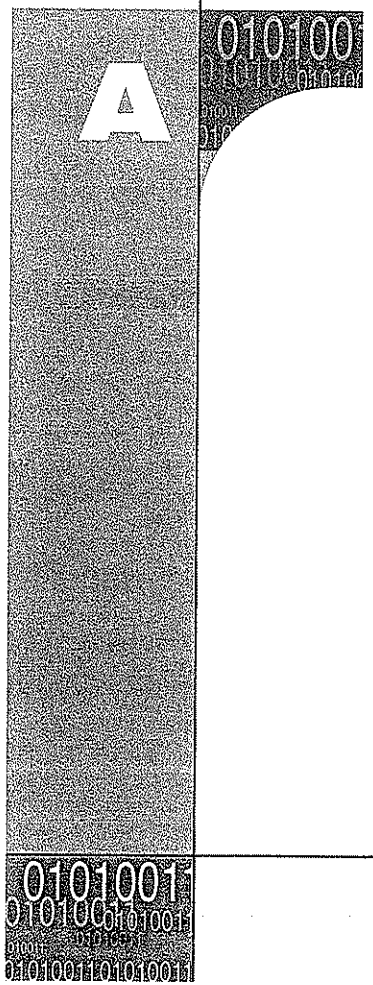
El presente (en 1996 escribíamos "el futuro") es la programación de aplicaciones distribuidas y paralelas, por ejemplo, sobre Internet y muchas veces en tiempo real (*on line*). Las aplicaciones con datos multimediales a distancia crecen en importancia (videoconferencia, realidad virtual, tratamiento de señales, procesamiento de imágenes en telemedicina, etc.)

Para el especialista informático se pierde la noción de su computadora local:

Su computadora puede ser una arquitectura enorme con miles de computadoras distribuidas en el mundo; sus datos están dispersos en estas múltiples máquinas e incluso parcialmente replicados en diferentes puntos; sus algoritmos pueden migrar ejecutándose en forma transparente en distintos procesadores,

Pero esencialmente el problema creativo a resolver sigue siendo crear el algoritmo *in the small* que realiza la solución del problema del mundo real, utilizando la computadora.

Desde ese punto de vista este texto trata de ser un pequeño aporte a la formación del lector, que se inicia en la programación de algoritmos.



Apéndice A

Visual Da Vinci

Visual Da Vinci

Hace algunos años, un grupo de investigadores del LIDI, desarrollaron la especificación de una máquina abstracta (el robot Lubo-I) y la de un lenguaje de programación asociado a la misma, ambas destinadas a la enseñanza de los conceptos introductorios de la programación estructurada (De Giusti; 1988, 1989).

Esas especificaciones fueron tomadas para el desarrollo de un entorno gráfico para el desarrollo y ejecución de programas orientado a la enseñanza inicial de programación estructurada, llamado Visual Da Vinci (Champredonde; 1997, 2000).

Esta herramienta permite escribir programas que definen el comportamiento de un robot virtual, así como verificar su validez sintáctica y ejecutarlos, por medio de un ambiente gráfico, sencillo, amigable e intuitivo.

Un programa determina qué debe hacer el robot, en qué orden y cuántas veces. En otras palabras, un programa determina el algoritmo a seguir por el robot, el cual puede incluir un conjunto limitado de acciones que son descritas más adelante.

El resultado de la ejecución de un programa se visualiza en forma gráfica de manera tal que el alumno realiza una verificación visual de la corrección del programa.

Los programas son escritos en un lenguaje específico determinado por su sintaxis y su semántica. Este lenguaje está compuesto por los elementos básicos indispensables que permiten transmitir los conceptos de programación estructurada de una forma simple y progresiva, sin desviar la atención en temas tangenciales, poniendo énfasis en la formación de un estilo de programación y arribando paulatinamente hacia un lenguaje convencional tradicionalmente utilizado para la enseñanza de programación estructurada, como lo es Pascal.

El robot

Visual Da Vinci es una herramienta para programar el comportamiento de un robot virtual por medio de la especificación de programas.

Este robot virtual puede circular por una ciudad también virtual. La ciudad está compuesta por 100 calles y 100 avenidas, numeradas entre 1 y 100. Las calles son horizontales y las avenidas son verticales. Como es natural, las intersecciones de las calles con las avenidas son las esquinas. Cada esquina es identificada por un par ordenado (av, ca) siendo av y ca números enteros entre 1 y 100.

Se dice que la parte superior de la ciudad es el norte, la parte inferior es el sur, la parte de la derecha es el este y la parte de la izquierda el oeste.

La Figura A.1 muestra la parte de la ciudad que entra en la ventana correspondiente.

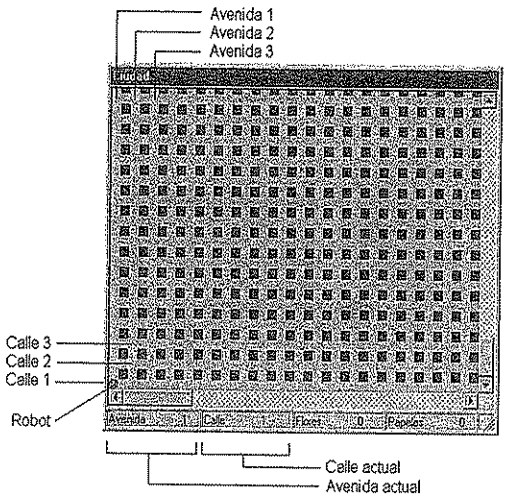


Figura A.1

El robot posee tres capacidades relacionadas a su posición dentro de la ciudad:

- Posicionamiento inicial: la posición inicial del robot en todo programa es la esquina (1,1) orientado hacia el norte. El posicionamiento inicial debe realizarse una y solo una vez en cada programa.
- Traslación: el robot puede desplazarse de la esquina en la que se encuentra a la siguiente de acuerdo a su orientación. Por ejemplo, si se encuentra en la esquina (5,10) orientado hacia el norte, puede trasladarse a la esquina (5,11). Si, en cambio, estuviese orientado hacia el este, se trasladaría a la esquina (6,10). La traslación del robot no tiene ningún efecto sobre la orientación del mismo. El programador debe considerar las situaciones en las que el robot se encuentre ubicado sobre un borde de la ciudad y orientado hacia fuera, ya que un movimiento del robot implicaría que se "caiga" de la ciudad o, en otras palabras, que se escape de los límites permitidos de la ciudad.
- Rotación: el robot puede cambiar su orientación por medio de giros de 90° en sentido horario. Por ejemplo, si el robot se encuentra orientado hacia el norte, puede rotar de forma tal de quedar orientado hacia el este, o si está orientado hacia el sur puede rotar para quedar orientado hacia el oeste. La rotación del robot no tiene ningún efecto sobre la posición del mismo.

En cada una de las esquinas de la ciudad puede haber una cierta cantidad de flores y una cierta cantidad de papeles (ver Figura A.2). Además el robot lleva consigo una bolsa de flores con una cierta cantidad de flores y una bolsa de papeles con una cierta cantidad de papeles.

Si en la esquina en la que está posicionado el robot hay flores, este puede tomarlas y ponerlas en su bolsa de flores. Si en la esquina en la que está posicionado el robot hay papeles, este puede tomarlos y ponerlos en su bolsa de papeles.

Por otro lado, si el robot tiene papeles en su bolsa de papeles, puede depositarlos en la esquina en la que se encuentra posicionado, y si tiene flores en su bolsa de flores, puede depositarlas en la esquina en la que se encuentra posicionado.

El robot tiene forma de saber si hay flores y/o papeles en la esquina en la que se encuentra, así como si hay flores y/o papeles en sus respectivas bolsas. Sin embargo, no tiene forma de saber cuántas son las flores ni cuántos los papeles.

Adicionalmente, en cada esquina de la ciudad puede haber un obstáculo que impide el paso del robot por esa esquina. El robot puede "ver" si en la esquina siguiente de acuerdo a su orientación hay un obstáculo o no, de forma tal que puede saber si el avanzar hacia la siguiente esquina ocasionará una colisión. Se dice que un obstáculo es simple cuando no se encuentran más obstáculos en las esquinas aledañas a la que se halla. Los obstáculos pueden formar "barreras" cuando se encuentran en esquinas alineadas y continuas.

En la Figura A.2 se muestra la forma en la que se representan las flores, los papeles y los obstáculos dentro de la ciudad.

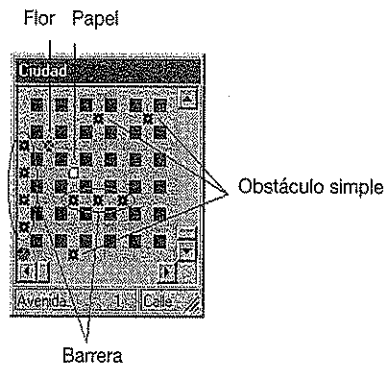


Figura A.2

Lenguaje de Programación

El comportamiento del robot se determina por medio de la especificación de un "programa". Esta especificación se realiza utilizando un lenguaje de programación que, como todo lenguaje, está definido por su sintaxis y su semántica.

Todo programa Da Vinci está compuesto por tres partes: encabezamiento, declaraciones y cuerpo.

El encabezamiento está compuesto por la palabra clave `programa` seguida de un identificador que determina el nombre del programa.

En la segunda parte, se declaran los subprogramas y las variables que utilizará el programa principal. El orden de estas declaraciones es importante. No se permiten declaraciones de variables anteriores a subprogramas, de forma tal que no existe la posibilidad de hacer referencia a variables globales desde un subprograma.

La declaración de subprogramas comienza con la palabra clave `procesos`, y termina donde se encuentra la palabra clave `variables` o bien la palabra clave `comenzar`.

Por otro lado, la declaración de las variables comienza con la palabra clave `variables` y termina donde se encuentra la palabra clave `comenzar`.

El cuerpo del programa principal es una secuencia de sentencias, delimitada por las palabras clave `comenzar` y `fin`. Los cuerpos de los subprogramas siguen la misma regla.

Variables

La palabra clave `variables` indica el comienzo de la sección de declaraciones de variables del programa o de un subprograma, la que llega hasta la palabra clave `comenzar` del cuerpo de la unidad correspondiente.

Cada variable tiene un nombre y un tipo. Los tipos posibles son solamente dos: `numero` y `boolean`.

El hecho de que el conjunto de tipos posibles sea reducido a solo dos elementos es una restricción importante del lenguaje. Sin embargo, el objetivo de este lenguaje es la enseñanza de programación estructurada y, por esta razón, se pone atención principalmente en los aspectos relacionados con los algoritmos y no tanto en los relacionados con los tipos y estructuras de datos.

Como en la mayoría de los lenguajes de programación, una variable puede ser asignada con un valor del tipo correspondiente, y puede intervenir en cualquier expresión.

El alcance y la visibilidad de una variable está limitada al cuerpo del programa o subprograma que la define.

Subprogramas

Los subprogramas, llamados procesos en Visual Da Vinci, tienen una estructura casi idéntica a la del programa principal.

Cada subprograma está dividido en las mismas tres partes que un programa, es decir, encabezamiento, declaraciones y cuerpo.

Las declaraciones de procesos locales y variables locales siguen exactamente las mismas reglas que las declaraciones del programa principal. Lo mismo sucede con el cuerpo del subprograma.

La única diferencia en la especificación de un subprograma y la del programa principal es que el primero puede definir parámetros formales en su encabezamiento.

Parámetros formales

El pasaje de parámetros en Visual Da Vinci es conceptualmente igual al del lenguaje Ada (USDOD, 1983). Fue elegido este modelo de pasaje de parámetros debido a que resulta más representativo que otros de los modos usuales en los lenguajes de programación convencionales.

Los parámetros formales de un subprograma son definidos entre paréntesis como parte del encabezamiento.

Cada parámetro formal tiene un nombre, un tipo y un modo.

El nombre de un parámetro lo identifica dentro del subprograma que lo define. Su alcance y su visibilidad son los mismos que los de una variable local.

Los tipos posibles de un parámetro son, al igual que para las variables, `numero` y `boolean`.

Los modos de pasaje de parámetro que se pueden utilizar son: entrada, salida y entrada/salida, indicados respectivamente por `E`, `S` y `ES`.

Un parámetro formal de entrada actúa como una constante local. Es decir, puede intervenir en cualquier expresión, pero no puede ser modificado; no puede estar a la izquierda de una asignación.

Por el contrario, la única operación permitida sobre un parámetro de salida es asignarle un valor. No puede participar en una expresión, solo puede aparecer a la izquierda de una asignación.

Un parámetro de entrada/salida es equivalente a una variable local. Puede aparecer en cualquier lugar que pueda hacerlo una variable.

Sentencias

Como ya se ha dicho, el cuerpo de cada módulo, ya sea programa o proceso, es una secuencia de sentencias.

El conjunto de las sentencias está dividido en tres subconjuntos: primitivas, sentencias simples y sentencias compuestas.

Las primitivas son aquellas instrucciones directamente ejecutables por el robot. Ellas son: `iniciar`, `derecha`, `mover`, `tomarFlor`, `tomarPapel`, `depositarFlor`, `depositarPapel`.

Las sentencias simples son la asignación y la invocación a procesos definidos por el usuario o a procesos del sistema (`Pos`, `Informar`).

La invocación a procesos que definen parámetros formales debe incluir los respectivos parámetros reales. La correspondencia entre parámetros formales y reales queda determinada por la posición de los mismos.

Las sentencias compuestas son las estructuras de control `mientras`, `repetir`, `si` y `si/sino`. Se denominan compuestas porque incluyen a una o más sentencias ya sean simples o compuestas.

El `mientras` es una estructura iterativa condicional. Contiene una expresión booleana y una secuencia de sentencias, la cual se ejecuta mientras el resultado de evaluar la condición es verdadero.

El `repetir` es también una estructura iterativa, pero incondicional. Está compuesta por una expresión aritmética y una secuencia de sentencias. El resultado de la evaluación de la expresión aritmética determina la cantidad de veces que se repite la ejecución del cuerpo del `repetir`.

La estructura `si` tiene una condición lógica y una o dos secuencias de sentencias. Si tiene dos, ambas se encuentran separadas por la palabra clave `sino`. La primer secuencia se ejecuta solo si se cumple la condición. Si la condición resulta falsa y el `si` tiene una segunda secuencia de sentencias (la secuencia del `sino`), entonces es esta la que se ejecuta.

Expresiones

Las expresiones que intervienen en un programa Visual Da Vinci pueden ser lógicas o aritméticas.

Se construyen relacionando variables definidas por el usuario, variables del sistema, valores lógicos y numéricos, y parámetros de entrada y entrada/salida, por medio de operadores lógicos, aritméticos y/o relacionales.

Un valor, una variable o un parámetro son también considerados expresiones.

Las variables del sistema son: `HayFlorEnLaEsquina`, `HayPapelEnLaEsquina`, `HayFlorEnLaBolsa`, `HayPapelEnLaBolsa`, `HayObstaculo`, `PosAv`, `PosCa`.

Los operadores aritméticos son: + (suma), - (resta), * (multiplicación) y / (división). Los booleanos: ~ (not), & (and) y | (or). Y los relacionales: = (igual), <> (distinto), > (mayor), < (menor), >= (mayor o igual) y <= (menor o igual).

Sintaxis

Aquí se describen las reglas sintácticas del lenguaje por medio de una notación al estilo BNF, en la que <X> indica que X es un símbolo no terminal, [X] que X no es obligatorio, {X} que pueden aparecer cero o más repeticiones de X, X|Y es la selección de uno de X o Y. Los símbolos en negrita son símbolos terminales. El símbolo "__" representa la indentación de dos espacios, obligatoria en muchos casos.

```
<Programa>::=
<Encabezamiento de programa>
[<Declaraciones>]
<Cuerpo>

<Encabezamiento de programa>::=
programa <Ident. de programa>

<Ident. programa> ::=
<Identificador>

<Identificador>::=
(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|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){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|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}

<Declaraciones>::=
[<Declaración de procesos>]
[<Declaración de variables>]

<Declaración de procesos>::=
procesos
__{<Declaración de un proceso>}

<Declaración de un proceso>::=
<Encabezamiento de proceso>
[<Declaraciones>]
<Cuerpo>

<Encabezamiento de proceso>::=
proceso <Ident.proceso>[(<Parámetros formales>)]

<Ident.proceso>::=
<Identificador>

<Parámetros formales>::=
{<Parámetro formal>, }<Parámetro formal>

<Parámetro formal>::=
<Parám. Formal de entrada>|
```

```
<Parám. Formal de salida>|
<Parám. Formal de entr./sal.>

<Parám. formal de entrada>::=
E <Ident. parám. formal entr.>: <Tipo>

<Ident. parám. formal entr.> ::=
<Identificador>

<Parám. formal de salida>::=
S <Ident. parám. formal sal.>: <Tipo>

<Ident. parám. formal sal.> ::=
<Identificador>

<Parám. formal de entr./sal.> ::=
ES <Ident. parám. formal entr./sal.>: <Tipo>

<Ident. parám. formal entr./sal.> ::=
<Identificador>

<Tipo>::=
numero|boolean

<Declaración de variables>::=
variables
__<Declaración de una variable>
__{<Declaración de una variable>}


<Declaración de una variable>
<Ident. variable>: <Tipo>

<Ident. variable>::=
<Identificador>

<Cuerpo>::=
comenzar
__{<Secuencia de sentencias>}
fin

<Secuencia de sentencias>::=
<Sentencia>
{<Sentencia>}

<Sentencia>::=
<Primitiva>|<Sentencia simple>|<Sentencia compuesta>

<Primitiva>::=
iniciar|mover|derecha|tomarFlor|tomarPapel|depositarFlor|deposi-
tarPapel
```



```
<Sentencia simple>::=
<Asignación>|<Invocación>

<Asignación>::=
(<Ident. variable>|
<Ident. parám. formal sal.>|
<Ident. parám formal entr./sal.>) := <Expresión>

<Expresión>::=
{(<Valor>|<Ident. variable>|
<Ident. var. sistema>|
<Ident. parám. formal entr.>|
<Ident. Parám  forma entr./sal.>|
(<Expresión>)|-<Expresión>|
~<Expresión>) <Operador>} (<Valor>|<Ident. variable>|
<Ident. var. sistema>| <Ident. parám. formal entr.>| <Ident. pa-
rám. formal entr./sal.>|
(<Expresión>)|-<Expresión>|~<Expresión>)

<Valor>::=
<Valor numérico>|<Valor booleano>

<Valor numérico>::=
0|1|2|3|4|5|6|7|8|9{0|1|2|3|4|5|6|7|8|9}
[.0|1|2|3|4|5|6|7|8|9{0|1|2|3|4|5|6|7|8|9}]

<Valor booleano>::=
V|F

<Ident. var. sistema>::=
PosAv|PosCa|HayFlorEnLaEsquina|
HayFlorEnLaBolsa|HayPapelEnLaEsquina|
HayPapelEnLaBolsa

<Operador>::=
<Operador aritmético>|<Operador booleano>| <Operador relacional>

<Operador aritmético>::=
+|-|*|/

<Operador booleano>::=
~|&||

<Operador relacional>::=
=|<>|<|>|<=|>=

<Invocación>::=
(<Ident. proceso>|<Ident. proc. sistema>) [<Parámetros reales>]

<Ident. proc. sistema>::=
Pos|Informar
```

```
<Parámetros reales>::=
<Parám. real entr.>|<Parám.realsal.>| <Parám. real entr./sal.>

<Parám. real entr.>::=
<Expresión>

<Parám. real sal.> ::=
<Ident. variable>|
<Ident. parám. formal sal.>| <Ident. parám. formal entr./sal.>

<Parám. real entr./sal.> ::=
<Ident. variable>|
<Ident. parám. formal entr./sal.>

<Sentencia compuesta>::=
<Selección>|<Iteración condicional>| <Iteración incondicional>

<Selección>::=
si <Expresión>
__<Secuencia de sentencias>
[sino
 __<Secuencia de sentencias>]

<Iteración condicional>::=
mientras <Expresión>
__<Secuencia de sentencias>]

<Iteración incondicional>::=
repetir <Expresión>
__<Secuencia de sentencias>
```

Semántica

`programa` Indica el comienzo de la especificación de un programa. Debe estar seguido por un identificador que da el nombre al programa.

`procesos` Indica el comienzo de la sección de declaraciones de procesos.

`variables` Indica el comienzo de la sección de declaraciones de variables.

`comenzar` Indica el comienzo de la secuencia de sentencias que componen el cuerpo del programa principal o de un proceso.

`fin` Indica el final de la secuencia de sentencias que componen el cuerpo del programa principal o de un proceso.

`proceso` Indica el comienzo de la definición de un proceso. Debe estar seguido de un identificador que da el nombre al proceso.

`E` Indicador de modo de pasaje de parámetro para parámetros de entrada.

`S` Indicador de modo de pasaje de parámetro para parámetros de salida.

`ES` Indicador de modo de pasaje de parámetro para parámetros de entrada/salida.

`numero` Nombre del conjunto de todos los valores numéricos.

`boolean` Nombre del conjunto de los valores lógicos.

`mientras` Indica el comienzo de una estructura de control iterativa condicional. Debe estar seguido por una condición (expresión booleana) y una secuencia de sentencias. Esta secuencia es ejecutada mientras se cumpla la condición. Esto significa que será ejecutada cero o más veces.

`repetir` Indica el comienzo de una estructura de control iterativa incondicional. Debe estar seguido por una expresión aritmética y una secuencia de sentencias. Esta secuencia es ejecutada una cantidad fija de veces, la cual es determinada por el resultado de la evaluación de la expresión.

`si` Indica el comienzo de una estructura de control condicional. Debe estar seguida de una condición (expresión booleana) y una secuencia de sentencias. Esta secuencia se ejecuta solamente si se cumple la condición. Puede estar acompañado por un `sino` y otra secuencia de sentencias. En este caso, cuando la condición del si no se cumple, se ejecuta esta última secuencia.

`iniciar` Instrucción primitiva que posiciona al robot en la esquina (1,1) orientado hacia el norte, e inicializa las variables del sistema.

`derecha` Instrucción primitiva que cambia la orientación del robot hacia el punto cardinal que difiere en 90° en sentido horario respecto de la orientación actual.

`mover` Instrucción primitiva que conduce al robot de la esquina en la que se encuentra a la siguiente, en la dirección hacia la cual está orientado. Es responsabilidad del programador que esta instrucción sea ejecutada solo cuando no haya un obstáculo en la esquina destino. En caso contrario, se producirá un error en tiempo de ejecución y el programa será abortado.

`tomarFlor` Instrucción primitiva que junta una flor de la esquina en la que se encuentra el robot y la pone en la bolsa de flores del mismo. Es

responsabilidad del programador que esta instrucción sea ejecutada solo cuando haya al menos una flor en dicha esquina. En caso contrario, se producirá un error en tiempo de ejecución y el programa será abortado.

`tomarPapel` Instrucción primitiva que junta un papel de la esquina en la que se encuentra el robot y lo pone en la bolsa de papeles del mismo. Es responsabilidad del programador que esta instrucción sea ejecutada solo cuando haya al menos un papel en dicha esquina. En caso contrario, se producirá un error en tiempo de ejecución y el programa será abortado.

`depositarFlor` Instrucción primitiva que saca una flor de la bolsa de flores del robot y la deposita en la esquina en la que este está posicionado. Es responsabilidad del programador que esta instrucción sea ejecutada solo cuando haya al menos una flor en dicha bolsa. En caso contrario se producirá un error en tiempo de ejecución y el programa será abortado.

`depositarPapel` Instrucción primitiva que saca un papel de la bolsa de papeles del robot y lo deposita en la esquina en la que este está posicionado. Es responsabilidad del programador que esta instrucción sea ejecutada solo cuando haya al menos un papel en dicha bolsa. En caso contrario se producirá un error en tiempo de ejecución y el programa será abortado.

`PosAv` Variable del sistema, cuyo valor es un número entero en el rango 1...100, que indica la avenida en la que el robot está actualmente posicionado. Su valor no puede ser modificado por el programador.

`PosCa` Variable del sistema, cuyo valor es un número entero en el rango 1...100, que indica la calle en la que el robot está actualmente posicionado. Su valor no puede ser modificado por el programador.

`HayFlorEnLaEsquina` Variable del sistema, cuyo valor es V si hay al menos una flor en la esquina en la que el robot está actualmente posicionado, o F en caso contrario. Su valor no puede ser modificado por el programador.

`HayPapelEnLaEsquina` Variable del sistema, cuyo valor es V si hay al menos un papel en la esquina en la que el robot está actualmente posicionado, o F en caso contrario. Su valor no puede ser modificado por el programador.

`HayFlorEnLaBolsa` Variable del sistema, cuyo valor es V si hay al menos una flor en la bolsa de flores del robot, o F en caso contrario. Su valor no puede ser modificado por el programador.

`HayPapelEnLaBolsa` Variable del sistema, cuyo valor es V si hay al menos un papel en la bolsa de papeles del robot, o F en caso contrario. Su valor no puede ser modificado por el programador.

`HayObstaculo` Variable del sistema, cuyo valor es V si hay un obstáculo en la esquina siguiente a la que está posicionado actualmente el robot, o F en caso contrario. Su valor no puede ser modificado por el programador.

`Pos` Proceso del sistema que recibe dos parámetros de entrada `Av` y `Ca`, cada uno de ellos en el rango 1...100, y posiciona al robot en la esquina determinada por el par (Av, Ca).

`Informar` Proceso del sistema que recibe una cantidad cualquiera de parámetros de entrada, y muestra en pantalla sus valores.

Estilo de programación

Se imponen ciertas reglas sintácticas adicionales (que no son muchas, pero sí estrictas), las cuales se ocupan únicamente de la indentación y el uso de mayúsculas y minúsculas.

Aunque pueden resultar algo incómodas para el programador, esas reglas intentan formar un buen estilo de codificación. Si el término "buen estilo" resultara discutible, en todo caso se podría decir "un estilo". Lo importante aquí es presentar al estudiante algunos criterios a tener en cuenta en el momento de crear su propio estilo.

El objetivo de un estilo de programación es mejorar en todo lo posible la legibilidad del código, de forma tal que resulte sencillo entenderlo, modificarlo, adaptarlo y reutilizarlo, ayudando así a maximizar la productividad y minimizar el costo de desarrollo y mantenimiento.

En el código textual se deben respetar las siguientes reglas de indentación:

- La palabra clave `programa` debe comenzar en la primera columna.
- Las palabras clave `procesos`, `variables`, `comenzar` y `fin` de un programa deben comenzar en la misma columna que la palabra clave `programa`.
- Las palabras clave `procesos`, `variables`, `comenzar` y `fin` de un proceso deben comenzar en la misma columna que la palabra clave `proceso`.
- La definición de un proceso debe comenzar dos columnas más a la derecha que la palabra clave `procesos` que indica el comienzo de la sección de declaraciones de procesos.
- Las declaraciones de variables deben comenzar dos columnas más a la derecha que la palabra clave `variables` que indica el comienzo de la sección de declaraciones de variables.

- Las sentencias de los cuerpos del programa y de los procesos deben comenzar dos columnas más a la derecha que las palabras clave `comenzar` y `fin` que delimitan a esos cuerpos.
- Las sentencias que pertenecen al cuerpo de una estructura de control deben comenzar dos columnas más a la derecha que la palabra clave que identifica a la estructura de control.
- La indentación es la única forma de indicar si una sentencia está dentro de una estructura de control o no. Esta forma de indentación está inspirada en la utilizada por el lenguaje OCCAM (CSA, 1990).

Por otro lado, todas las palabras claves definidas por el lenguaje, así como las primitivas, las estructuras de control y los tipos, deben ser escritas siempre con letras minúsculas, excepto que su nombre esté compuesto por más de una palabra, en cuyo caso, de la segunda palabra en adelante, cada una comienza con mayúscula. Por ejemplo, `iniciar`, `tomarFlor`, `mientras`, `numero`.

Por el contrario, las variables del sistemas y los procesos del sistema deben comenzar cada palabra que compone su nombre con una letra mayúscula y las demás minúsculas. Por ejemplo, `PosAv`, `HayFlorEnLaBolsa`, `Pos`, `Informar`.

Los indicadores de modo de pasaje de parámetros (`E`, `S`, `ES`) van siempre en mayúsculas. Se sugiere, además, que los identificadores definidos por el usuario sigan las convenciones de las variables y procesos del sistema. El usuario tiene absoluta libertad en el uso de mayúsculas y minúsculas cuando define un identificador. Sin embargo, cada vez que se haga referencia a un identificador definido por el usuario este debe ser escrito exactamente igual. En otras palabras, Visual Da Vinci es *case sensitive*, es decir, sensible a si las letras son mayúsculas o minúsculas. Con este criterio, los identificadores `cantidadFlores`, `CantidadFlores`, `cantidadflores` y `Cantidadflores` son todos distintos. Esta sensibilidad al tipo de letra es la misma que la del lenguaje C (Kernighan, 1988).

También se recomienda la utilización frecuente de comentarios lógicos que aclaren ciertos algoritmos, indiquen la utilidad de las variables y procesos utilizados, el objetivo de los parámetros, expliquen los motivos de algún compromiso tomado, etc.

Ejemplo

En el siguiente ejemplo se recorren 10 cuadrados concéntricos, comenzando en el cuadrado determinado por las esquinas (1,1) y (20,20) hasta el cuadrado determinado por las esquinas (10,10) y (11,11), juntando y contando todas las flores que se encuentren en el camino. Al finalizar el recorrido, se informa la cantidad de flores encontradas.

```
{ Recorre 10 cuadrados concéntricos, comenzando por el   }
{ cuadrado determinado por las esquinas (1, 1) y         }
{ (20, 20), hasta llegar al determinado por las esquinas }
```



```
{ (10, 10) y (11, 11), juntando y contando todas las   }
{ flores que encuentra en su camino. Al finalizar el   }
{ recorrido informa la cantidad de flores juntadas.    }

programa Concentricos

procesos
  { En los siguientes dos procesos se utiliza un       }
  { parámetro de entrada/salida. Ese parámetro podría  }
  { haber sido de solo salida.                         }
  { Pero así se evita tener que utilizar variables     }
  { locales auxiliares.                                }

  proceso HacerLado(E Largo: numero; ES CantFlores: numero)
    { Se encarga de recorrer Largo cuadras en la       }
    { dirección hacia la cual está orientado, juntando }
    { todas las flores que encuentra en su camino. Por }
    { cada flor que junta incrementa en 1 el parámetro }
    { de entrada/salida CantFlores.                    }
  comenzar
    repetir Largo
      mover
      mientras HayFlorEnLaEsquina
        tomarFlor
        CantFlores := CantFlores + 1
  fin

  proceso HacerCuadrado(E Tamano: numero; ES Flores: numero)
    { Se encarga de recorrer un cuadrado de lado       }
    { Tamano, juntando las flores que encuentra en su  }
    { camino.                                          }
    { Por cada flor que junta incrementa en 1 el       }
    { parámetro de entrada/salida Flores.              }
  comenzar
    repetir 4
      HacerLado(Tamano, Flores)
      derecha
  fin

variables
  Lado: numero
  Flores: numero

comenzar
  iniciar
  Lado := 19
  Flores := 0
  repetir 10
    HacerCuadrado(Lado, Flores)
    { El siguiente cuadrado a recorrer tiene 2 cuadras }
```

```
      { menos de lado que el cuadrado recientemente        }
      { recorrido.                                         }
      Lado := Lado - 2
      { El próximo cuadrado comienza una avenida más a la }
      { derecha y una calle más arriba que el              }
      { recientemente recorrido.                           }
      Pos(PosAv + 1, PosCa + 1)
   Informar(Flores)
fin
```

Ambiente de desarrollo

El ambiente de desarrollo de Visual Da Vinci puede obtenerse en http://lidi.info.unlp.edu.ar/davinci.zip. El mismo está compuesto por cinco ventanas: la ventana principal, el editor de diagramas visuales, el editor de código textual, la ciudad y el inspector de variables del sistema.

La ventana principal permite la manipulación global de archivos de programa, incluyendo las opciones de verificación de sintaxis, ejecución y depuración.

Cuenta con un menú y dos barras de botones. Todas las acciones que se pueden realizar por medio de los botones de la primera barra (situada en el lado izquierdo) pueden ser también realizadas utilizando las opciones correspondientes del menú.

En el editor de código se escriben los programas que, luego de compilados, se pueden ejecutar. El resultado de ejecutar un programa se puede visualizar en la ventana de la ciudad, en donde se ve al robot realizando el recorrido y las acciones indicadas en el programa.

Por medio del menú Opciones, que se encuentra en la ventana principal, se puede seleccionar la cantidad de flores, papeles y obstáculos que serán puestos automáticamente antes de la ejecución de cada programa. También se puede seleccionar la cantidad de flores y papeles que inicialmente lleva el robot en sus respectivas bolsas.

En la ventana más pequeña, inicialmente situada sobre la izquierda de la pantalla, se muestra la posición y la dirección actuales del robot, así como las cantidades de flores y papeles que tiene el robot en sus bolsas.

Los programas pueden ser desarrollados también en forma visual, por medio de la confección de un diagrama. Para ello se utiliza el editor de diagramas. A medida que se confecciona un diagrama automática y simultáneamente se genera el código textual correspondiente y se muestra en el editor de código.

El camino inverso no se cumple. Es decir, que las modificaciones que se realicen en el editor de código no se reflejarán en el editor de diagramas.

Conclusiones

Este anexo presenta un ambiente de desarrollo de programas para el manejo de un robot virtual y el lenguaje con el cual escribir tales programas.

Tanto el lenguaje como el entorno gráfico de desarrollo fueron específicamente diseñados y desarrollados para la enseñanza inicial de programación estructurada.

B

Apéndice B

Definiciones de Ingeniería de Software

Definiciones de Ingeniería de Software

Definición

La Ingeniería de Software es el área de la ciencia Informática que trata el análisis, diseño e implementación de sistemas de software (Ghezzi, 1991).

Definición

Ingeniería de Software es el estudio de los principios y metodologías para el desarrollo y mantenimiento de sistemas de software (Zelkovitz, 1979).

Definición

La Ingeniería de Software es la disciplina tecnológica y administrativa dedicada a la producción sistemática de productos de programación, que son desarrollados y modificados a tiempo y dentro de un presupuesto definido (Fairley, 1987).

Definición

Ingeniería de Software es la aplicación práctica del conocimiento científico en el diseño y construcción de programas de computadora y la documentación asociada, requerida para desarrollar, operar y mantenerlos. Se conoce también como desarrollo o producción de software (Bohem).

Definición

La Ingeniería de Software trata del establecimiento de los principios y métodos de la Ingeniería a fin de obtener software de modo rentable, que sea confiable y trabaje en máquinas reales (Bauer, 1972).

Definición

Ingeniería de Software es la aplicación de un enfoque sistemático, disciplinado y cuantificable del desarrollo, operación y mantenimiento del software. (IEEE, 1993).

Definición

La Ingeniería de Software es una actividad industrial que requiere un enfoque disciplinado que abarca tres elementos claves: métodos, herramientas y procedimientos. Los métodos cubren todas las etapas del ciclo de vida del producto software; las herramientas suministran un soporte automático para los métodos y los procedimientos dan consistencia y coherencia a la aplicación de métodos y herramientas. (Arainty, 1996).

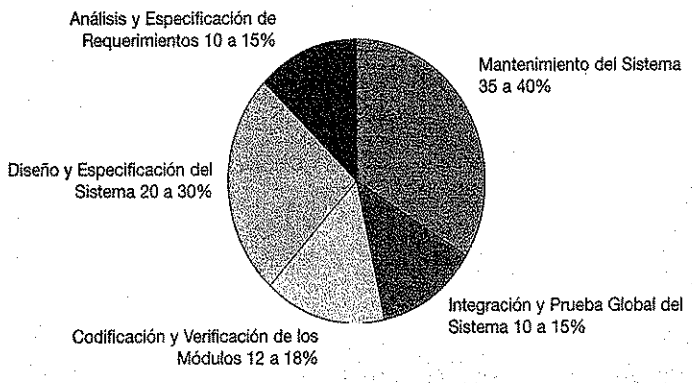
C

Apéndice C

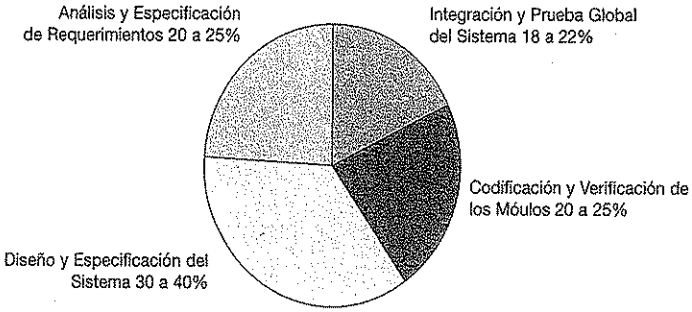
Análisis de costos relativos de cada etapa del ciclo de vida de un sistema de software

Análisis de costos relativos de cada etapa del ciclo de vida de un sistema de software

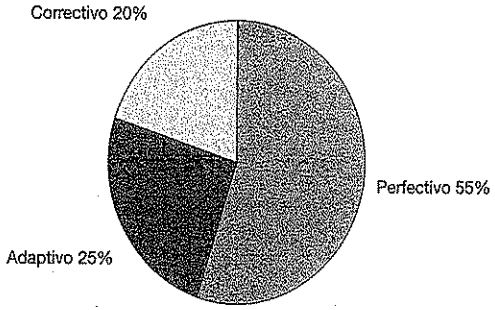
Costos relativos de las etapas del ciclo de vida



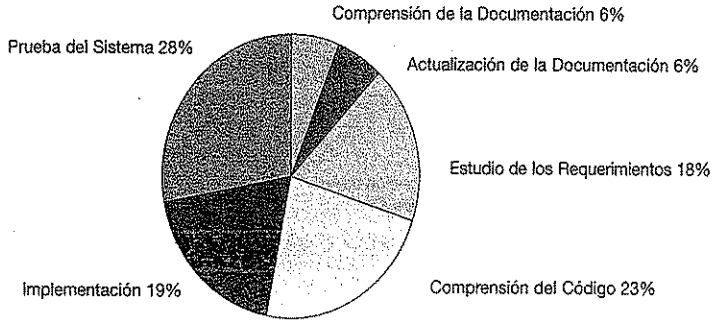
Costos relativos de las etapas del desarrollo de un sistema (sin mantenimiento)



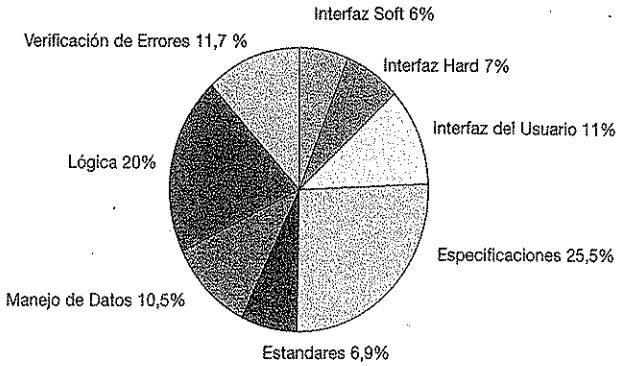
Costos relativos de los diferentes tipos de mantenimiento



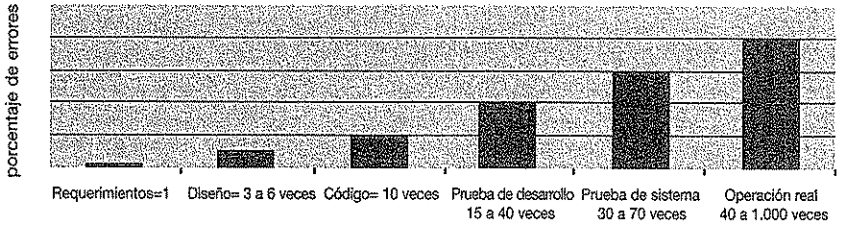
Costos relativos de los diferentes tipos de mantenimiento



Causas de defectos y su origen para un grupo de proyectos de software



Costo relativo de corregir un error



BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

Índice analítico

A

Abstracción, 9, 370, 373
Abstracción de datos, 121, 150, 276
Acceder a un elemento en la lista, 257
Administración de memoria, 200
Agregar un elemento a la lista, 255
Alcance de los datos, 48, 51
Algoritmo, 16,19, 23, 24, 281-282
Algoritmo de *Brute Force*, 315
Algoritmos de ordenación, 220
Algoritmos recursivos, 262, 271
Almacenamiento secundario, 328
Análisis de Algoritmos, 199, 201, 208
Análisis de algoritmos no numéricos, 314
Análisis de caso promedio, 293
Análisis del problema, 19
Árbol binario, 261
Árboles, 249, 259-262
Árboles balanceados de búsqueda, 356
Archivos, 329
  abrir, 330, 335
  acceso, 331
  actualización, 341
  administración, 329
  algoritmos clásicos, 338
  agregado de datos, 340
  secuencial, 331-332
  borrado, 354-355
  cerrar, 330, 335
  corte de control, 346
  directo, 331-332
  estratégias de Trabajo, 355
  eliminar elementos, ver borrado físico, 354
  fin de archivo (EOF), 331, 337
  lectura y/o escritura, 331, 335
  lógico, 355
  merge, ver Unión de archivos
    Modificación, 338
    Union de archivos, 349
  operaciones básicas, 330
  organización, 331,
  organización en Pascal, 333
Arreglo, 157
  bidimensional, 167
  de una dimensión, 157
  lineal, 157
Arreglos como parámetros, 176
Arreglos N-dimensionales, 174
Autómata, 17

B

Bases de datos, 355, 358
Borrado, 309
Borrar un elemento de la lista, 254
Buffers, 332
Bus de comunicación, 6
Búsqueda, 260-261
Búsqueda Binaria, 214-215, 260-261
Búsqueda Lineal, 212

C

Caso base, 187
Caso degenerado, 187-188-191-195
Ciclo de vida del software, 364
Clasificación de algoritmos, 292
Codificación y Verificación de los módulos, 365
Colas, 140, 250
  solución estática, 311, 312
  solución dinámica, 313, 314
Comentario, 18
Componentes, 379, 381
Computadora, 4
Constante, 17
Constantes, 71

asignaciones, 73
declaraciones, 71
Corrección de Algoritmos, 23, 38

D

Dato, 7
Datos dinámicos, 249
Datos Locales y Datos Globales, 49
Decisión, 28
Delphi
propiedades
eventos más comunes, 399
name, 398
text o *Caption*, 398
visible o *Enabled* 398
componentes
edit, 400
label, 400
maskEdit, 400
procesos
messageDig, 401
showMessage, 400
funciones de conversión
intToStr, 403
floatToStr, 403
strToFLoat, 403
strToInt, 403
Descomposición de problemas, 16, 41
Digital, 4
Diseño de una solución, 19
Documentación, 23

E

Editor de Diagramas de Visual Da Vinci, 379, 381
Eficiencia, 23, 40, 199, 371
Eficiencia en algoritmos recursivos, 208
Encapsulamiento de datos, 279
Entrada/salida, 6
Escritura de programas,19
Especificación de algoritmos, 19
Especificación, 13, 121
Estructuras de Control, 23
Estructuras de datos, 121, 122, 200, 249-250
compuestas, 122, 156
dinámica, 123, 156
estática, 123, 156
heterogénea, 122, 156
homogénea, 122, 156
lineales, 250
no lineales, 262
simples, 156
Etapas en la resolución de problemas, 19
Evento, 378, 390
Excepción, 404

F

Fibonacci, 297, 298
Formulario, 383
Fractal, 300, 301

G

Grafos, 249, 262, 266-271
dirigido, 264
no dirigido, 263

H

Hardware, 18, 156, 370

I

Identificadores, 80
Importancia de la Documentación, 38
Importancia de la Modularización, 42
Impresión, 306
Índice (de un arreglo), 157, 158, 167, 168
Índices, 355
Informática, 4, 364, 368, 370, 372
Ingeniería de Software, 364, 367, 370, 372-373
Inicialización, 303
Insertar, 304, 305
Inspector de Objetos, 381, 383, 390
Instrucción, 10, 369
Instrucciones, 24
Intercalar un elemento en la lista, 256
Iteración, 33

L

Lenguajes de programación, 14
Lenguajes fuertemente tipados, 72
Lenguaje Visual, 379
Lista, 249-258
circulares, 258
doblemente enlazadas, 258

M

Manejador de excepción, 404
Matriz, 167, 266
Memoria central, 6
de datos, 18,
principal, 328
secundaria, 328
volátil, 328
Método, 391
Método de Inserción, 225, 229
Método de Intercambio ó Burbuja, 223
Método de ordenación eficientes, 236
Método de Selección, 221
Modelo, 364, 368, 369
Modelización, 9
Modularización, 16
Módulo, 16
Módulos, 102
alcance, 103

O

Objeto, 388
Operaciones con listas, 254
Ordenación por Índice, 230
Orden de precedencia, 67
Overflow, 67

P

Paleta de Componentes, 381-382
Paradigma de programación, 363, 369
Parámetros, 52, 107
actuales, 107
correspondencia, 107
formales, 107
Pasaje de parámetros, 108
por resultado, 109
por referencia, 109
por valor, 108
por valor resultado, 109
Pertenece, 307, 308
Pilas,132, 250, 262
Pila de activación, 192-193
Poscondición, 11
Precondición, 11, 166
Procedimientos, 105
Programa, 6, 18, 281-282
Programación
funcional, 369
imperativa, 378
lógica, 369
orientada a eventos, 370, 378
procedural, 378
Programming in the small, 363, 368
Propiedad, 389
de lectura exclusiva, 389
de escritura exclusiva, 389
Propiedades de un objeto, 382
Proyecto, 379, 391
Punteros, 251-252

R

Recursividad, 245, 186-195
Refinamientos sucesivos, 363, 369
Registro, 123
Regla de cálculo, 298, 299
Relaciones básicas de recursión, 294
Repetición, 25
Representación de grafos, 266
Reusabilidad, 276, 287-288
Rutina, 102

S

Secuencia, 24
Sincrónico, 4
Sintaxis, 134, 142
Software, 18
Soluciones recursivas y no recursivas, 296
Soluciones rec. y no rec. En manejo de árboles, 302
Soluciones iterativas, 190
Sorting by Merging, 236
Subárbol, 260-261
Subprogramas, 102

T

TAD ver Tipo Abstracto de Datos
Tensor, 174
Tiempo de ejecución, 200,204
Tipo Abstracto de Datos, 80
Cuerpo de un TAD, 279
Definición, 277, 282
Genéricos, 284
Implementación, 278
Introducción, 275
Pilas y Colas como TAD's, 282
Tipo de dato, 79
carácter, 69
compuesto, 121
compuestos indexados, 155
conjunto, 87
declaración de tipos, 80
definidos por el usuario, 83
entero, 65
enumerativos, 83
lógico, 68
numérico, 64
ordenados discretamente, 77
ordinales, 77
real, 65
recursivos, 186
string, 91
subrango, 86
Tipo definido por el usuario, 121, 149
Top-Down, 102, 186
Tratamiento de *string*, 314
Try-except, 405

U

Underflow, 67
Unidad Central de Procesamiento, 6

V

Variable, 17
Variables, 71
asignaciones, 73
declaraciones, 71
locales, 114
globales, 114
Vector, 157
Verificación, 19
Visualización, 306

W

With, 129

COMPRA FACULTAD
$
Fecha 2004
Inv. E. Inv. B 1546

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

Pearson Educación

Visítenos en: www.pearsonedlatino.com

Argentina
Av. Regimiento de Patricios 1959
(C1266AAF) Buenos Aires
Argentina
Tel. (54-11) 4309-6100
Fax (54-11) 4309-6199
E-mail: info@pearsoned.com.ar

América Central Panamá
Barrio La Guaria, Moravia
75 metros norte
Del Portón Norte del Club La Guaria
San José, Costa Rica
Tel. (506) 235 72 76
Fax. (506) 297 28 52
E-mail: envwong@racsa.co.cr

Brasil
Rua Emilio Goeldi 747, Lapa
(05065-110) San Pablo, SP
Brasil
Tel. (5511) 861-0201
Fax (5511) 861-0654

Caribe
Monte Mall, 2do piso, suite 21-B
Av. Muñoz Rivera 652
Hato Rey
Puerto Rico 00918-4261
Tel. (787) 751-4830
Fax (787) 751-1677
E-mail: awlcarib@caribe.net

Chile
Av. Manuel Montt 1452
Providencia
Santiago, Chile
Tel. (562) 269 2089
Fax (562) 274 6158
E-mail: infopear@pearsoned.cl

Colombia
Carrera 68 #22-55
Santa Fe de Bogotá, DC
Colombia
Tel. (571) 405-9300
Fax (571) 405-9330

España
Núñez de Balboa 120
(28006) Madrid
España
Tel. (3491) 590-3432
Fax (3491) 590-3448

Estados Unidos
One Lake Street
Upper Saddle River
NJ 07458
Tel.(201) 236-7000
Fax: (201) 236-3400

México
Calle 4 N° 25, 2do piso
Fracc. Industrial Alce Blanco
(53370) Naucalpán de Juárez
Estado de México
Tel. (525) 387 07 00
Fax. (525) 387 08 11

Uruguay
Casa Juana de América
Av. 8 de Octubre 3061
(11600), Montevideo
Uruguay
Tel./fax (5982) 486-1617