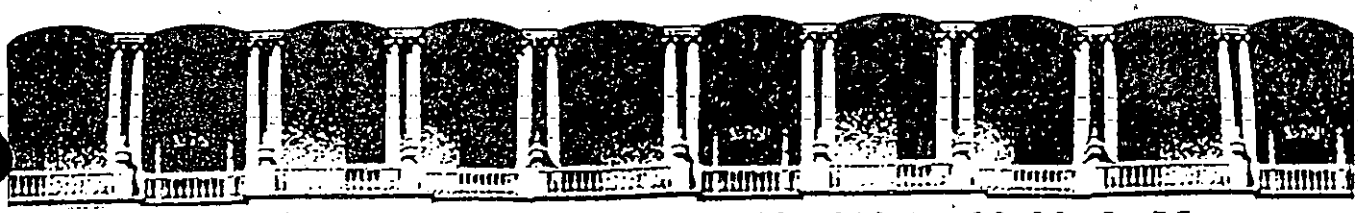
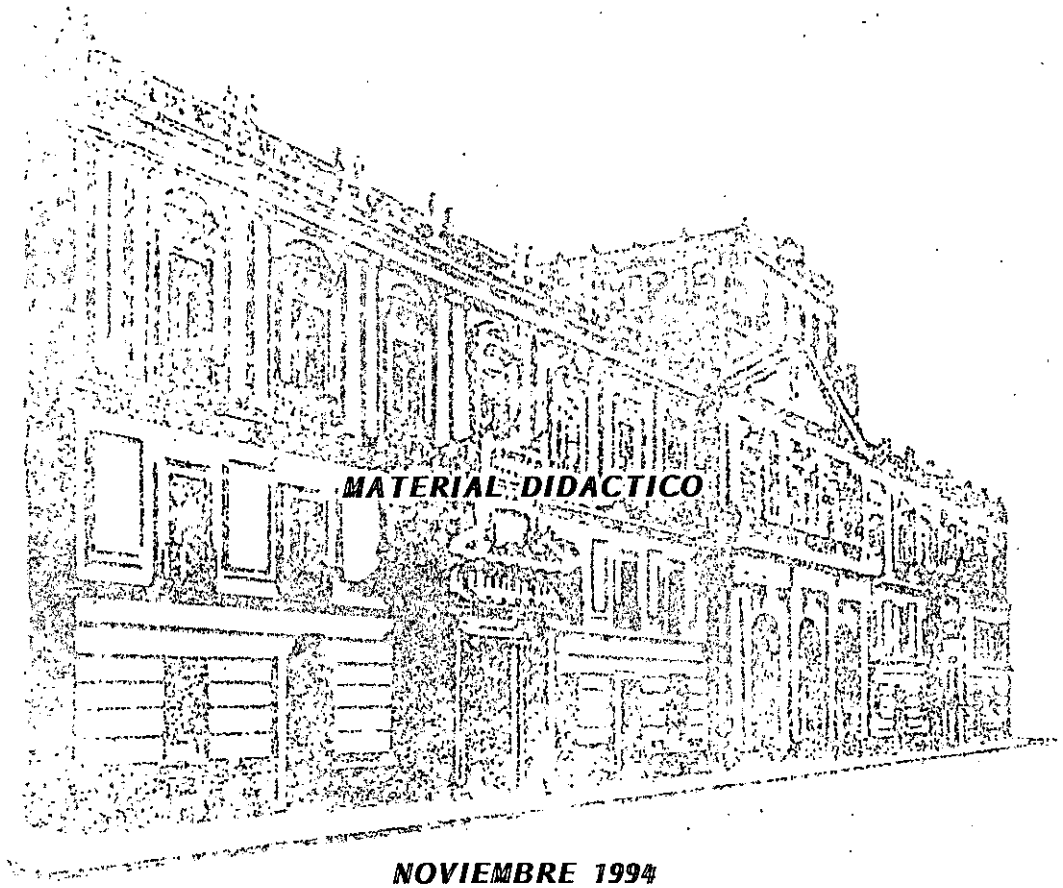


FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

INTRODUCCION A LA COMPUTACION

Y PROGRAMACION ELECTRONICA



MATERIAL DIDACTICO

NOVIEMBRE 1994

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. Tel.: 521-40-20 Apdo. Postal M-2285

Contenido

Prólogo a la segunda edición

Es grato poder retomar el material de la primera edición de este libro y trabajar en su actualización, y es grato por varias razones: la necesidad misma de hacerlo significa que el texto está cumpliendo su función en las aulas y, además, ofrece la oportunidad de acompañar de nuevo al lector en este viaje por las ciencias de la computación y los desarrollos tecnológicos que la enmarcan, lo que a su vez permite mostrar parte de los últimos avances en esta área tan dinámica.

En los cuatro años que han transcurrido desde la primera edición (y sus tres reimpresiones) ha habido algunos cambios en el entorno computacional, entre los que sobresalen el nivel de uso y la práctica ubicuidad de las computadoras personales y sus sistemas, lo que ha abierto nuevas alternativas de utilización y ha ampliado los horizontes de desarrollo de software. La intención es que esta nueva edición refleje, en la medida de lo posible, estos cambios.

Las modificaciones del material para esta segunda edición atañen tanto a la forma como al contenido, con el propósito de actualizar el texto en su totalidad. Entre los cambios de contenido está el que resulta de considerar el ya mencionado impacto de las computadoras personales; además se incluye una buena cantidad de bibliografía actualizada al final de cada capítulo, y se considera la existencia de nuevos lenguajes de programación y de nuevos desarrollos de inteligencia artificial. De los cambios de forma resaltan dos: la inclusión de ejercicios al final de cada capítulo y la división del capítulo 5 de la primera edición en dos partes (Caps. 5 y 6). Los ejercicios tienen el objetivo de que el lector disponga de material apropiado para poner en práctica los conocimientos adquiridos. Asimismo, se han incluido ejercicios de reflexión sobre los conceptos tratados en cada capítulo, porque en la concepción original de este libro la exposición conceptual de los diversos temas ocupaba —y ocupa— un lugar preponderante, no sólo desde un punto de vista taxo-

nómico, sino destacando el lugar que mantienen dentro de un esquema lógico necesario. Es decir, seguimos creyendo firmemente en el enfoque académico cuyo principio es que el alumno "invente" los conceptos conforme los requiera a lo largo del desarrollo de un tema, y no que sólo sea espectador del conocimiento. En el capítulo 4 se explica con más detalle este método, que hemos llamado "genético".

Este es un resumen, capítulo por capítulo, de los contenidos y cambios de esta nueva edición:

Capítulo 1: Se añadió una primera sección acerca de lo que he llamado la "razón de ser" de la computación y las computadoras, para explicar con mayor detalle los conceptos filosóficos en que se sustenta esta ciencia, que no puede ser, ni es, ajena a consideraciones humanistas.

Se expandió y puso al día la sección donde se describen las generaciones en que suele dividirse la historia del desarrollo de las computadoras.

Se dedica una sección completa a las microcomputadoras, así como un anexo en el que se describe parte de la terminología que suele emplearse al referirse a estos equipos.

Capítulo 2: Esta sección del texto prácticamente no tuvo modificaciones, pues la filosofía de funcionamiento de las computadoras no ha variado sustancialmente. Se incluyeron ejercicios y bibliografía.

Capítulo 3: Se actualizaron las referencias a los nuevos microprocesadores de 16 y 32 bits.

Se añadió un anexo en el que se describe el estándar internacional ISO/OSI para redes de computadoras.

Capítulo 4: Se añadió una sección para describir los editores de texto, las bases de datos y las hojas electrónicas de cálculo.

Se escribió una sección sobre inteligencia artificial y sistemas expertos.

Se añadió un anexo que describe el proceso de creación de una computadora de propósito especial, diseñada en México.

Capítulo 5: Este nuevo capítulo es una ampliación de la primera parte de aquel de la edición anterior, y aquí recibe el título de "computabilidad".

Se escribió una sección sobre gramáticas y lenguajes formales, que puede servir como introducción muy somera a ese tema tan importante. Se incluye un anexo sobre la historia de la lógica matemática y algunas consideraciones y bibliografía sobre filosofía de las matemáticas.

Capítulo 6: En la primera edición ésta era la segunda parte del capítulo 5, que aquí se ha separado en un capítulo completo, dada su importancia.

Se colocó en este capítulo el anexo sobre lenguajes de programación, que ahora tiene comentarios sobre 15 lenguajes diferentes.

Capítulo 7: Se ampliaron los conceptos sobre documentación y se añadió bibliografía sobre prueba de programas. Este era el capítulo 6 de la primera edición.

Al anexo que contiene programas codificados en diversos lenguajes de programación, que ya existía en la primera edición, se añadieron ejemplos en Forth, LISP, Modula-2 y Prolog, por lo que ahora incluye muestras de nueve diferentes lenguajes.

Se escribió además un breve anexo sobre lógica proposicional, porque se requiere para las labores de programación.

Capítulo 8: Lo que era el capítulo 7 de la primera edición tampoco tuvo modificaciones mayores; sólo adecuaciones en alguos puntos.

Se incluyeron ejercicios y bibliografía.

Apéndice A: Esto no existía en la primera edición. Se escribió un resumen de las principales características del sistema operativo Unix, enfocado fundamentalmente a mostrar sus principios de funcionamiento a la luz del modelo de sistemas operativos descrito en el capítulo 4. Este apéndice y el siguiente se deben a que tanto Unix como C se han convertido en virtuales estándares en un amplio segmento del quehacer computacional. Se incluyen referencias bibliográficas.

Apéndice B: Este apéndice, que tampoco existía en la primera edición, describe el lenguaje de programación C, en el que está escrito Unix, y hace algunas comparaciones contra Pascal y FORTRAN, los otros dos lenguajes manejados en este texto.
Se incluyen también referencias bibliográficas.

Al final de cada capítulo hay ahora una lista de las palabras y conceptos principales que se definieron en esa sección, con el objeto de que el lector pueda estar seguro de que los conoce, y pueda decidir si ya es capaz de decir algo interesante o significativo acerca de cada uno.

Se ha ampliado mucho, y puesto al día, la bibliografía comentada al final de cada capítulo. Están incluidos bastantes libros cuyo nivel rebasa ampliamente el carácter introductorio de este texto, pero su presencia responde más bien al deseo de que el lector tenga acceso a breves comentarios sobre una buena parte de los libros de computación en uso actual. Luego aparece un resumen de todos los libros y artículos mencionados en el texto, para efectos de facilitar y centralizar las referencias bibliográficas.

Esperamos que estos cambios cumplan la función de conservar el texto actualizado, pero cabe recordar al lector que este libro está constituido primordialmente por temas que no pierden su vigencia con rapidez, puesto que se trata de cuestiones conceptuales que están en la base misma de la computación y la computabilidad, y que son inmunes, hasta cierto grado, al acelerado ritmo de avance tecnológico. Esto puede decirse, por ejemplo, del capítulo 2, al que sólo se han hecho cambios menores, porque el modelo de computación allí descrito sigue siendo la base de funcionamiento de la mayoría de los equipos en existencia.

Aprecio a la Sra. María Fé Mojica S. Supervisora de Producción de McGraw-Hill, su atención y esfuerzo en la etapa final de este libro.

Agradezco de nuevo al Dr. Luis Legarreta sus comentarios y correcciones al contenido del texto, los cuales amablemente me hizo llegar en su oportunidad. Recibí también ayuda y comentarios importantes del maestro Luis Castro, de Jaime Huesca y de José Roberto Aguilar. Una vez más mis compañeros de oficina del Grupo Micrológica colaboraron ampliamente en esta labor, tanto con sus comentarios como con su apoyo. Laura Koestinger revisó la redacción de esta segunda edición de manera amistosa y desinteresada, cosa que estimo.

Prólogo a la primera edición

El presente texto tiene como propósito estudiar las técnicas elementales que intervienen en la moderna programación de computadoras digitales. Para esto es necesario tener una idea clara de su operación, que permita un entendimiento profundo de sus capacidades, limitaciones y potencialidad.

Puesto que las computadoras serán la herramienta para crear programas y sistemas de aplicación, es indispensable que el estudiante tenga una concepción clara de los principios del funcionamiento de estas máquinas, antes de aprender la metodología de programación estructurada que se presenta en este libro.

Así, el curso se ha dividido en dos partes; en la primera se explica qué es y cómo funciona un equipo de cómputo para luego, en la segunda, aprender a programar una computadora.

Por otro lado, hemos renunciado un tanto a los métodos tradicionales de exposición académica, en favor de un sistema que implica la participación integral del estudiante y pide de éste la exploración activa de los conceptos relevantes, en un proceso en el que se incita de continuo a la reflexión y el cuestionamiento lógico.

Este trabajo está dirigido a no especialistas en computación o ingeniería, y no presupone sino un conjunto elemental de conocimientos generales sobre matemáticas y lógica. Se parte de la idea de que primero es necesario entender intuitivamente un concepto para después poder profundizar en él. El desarrollo completo de lo que aquí se expone, tema de estudios más especializados que tendrán lugar en cursos posteriores, se encuentra ampliamente documentado en la bibliografía computacional.

Primera parte

Iniciamos con un breve repaso de la historia de las computadoras, desde los orígenes de los primitivos instrumentos de cálculo digital, hasta los actuales

equipos electrónicos. Se presenta también un análisis de los conceptos de fenómenos digitales y analógicos.

En el segundo capítulo se explica la idea central de las modernas computadoras, en términos de su funcionamiento interno. Se analiza el modelo de von Neumann, y se proporcionan ejemplos de su uso y funcionamiento, para después hacer un estudio descriptivo, en el capítulo 3, del sistema de cómputo integrado, es decir, del conjunto de elementos físicos y lógicos que lo configuran.

En el tercer capítulo se estudian las características más importantes de los componentes de una computadora, y se dan ideas acerca de su funcionamiento, sin entrar nunca en niveles operativos o que requieran conocimientos especializados. Se considera el concepto de procesamiento electrónico de datos y el de informática, y se describe una sesión normal en una terminal de computadora. Asimismo se habla de la organización de un centro de cómputo y sus funciones.

El capítulo 4 estudia, a título conceptual, los componentes lógicos de un sistema de cómputo, en contraposición con los elementos físicos o electrónicos recién descritos. A partir de la construcción conceptual, paso a paso, de un sistema de computación digital, se estudia la constitución de lo que se conoce como programación de sistemas; esto es, de los sistemas que, de alguna manera implícitos en una computadora, le permiten funcionar eficientemente y atender a sus usuarios. Aquí se tratan los fundamentos de los ensambladores, traductores de código y compiladores. La sección termina con un análisis somero acerca de la necesidad de los sistemas operativos, complementado con una descripción global de ellos.

Es importante aclarar que el método empleado hace hincapié en la comprensión de los conceptos y problemas por medio de intentos de solución, y no a través de una simple descripción. Por ello es que, cuando la extensión del tema lo permite, se prefiere mostrar el "interior" de un sistema (o la filosofía de su diseño), en lugar de sólo decir cómo se comporta.

En cierto sentido, el objetivo general de esta primera parte es explicar los modos de funcionamiento de un sistema de cómputo, con el fin de "desacralizar" a las computadoras; es decir, mostrar que estas máquinas son un producto social a nuestro alcance, y que debemos adquirir la capacidad de manejarlas y entenderlas para poder integrarlas adecuadamente a nuestros proyectos de desarrollo tanto sociales como individuales. Países como el nuestro deben hacer suyo este tipo de tecnología y, por ende, todo esfuerzo serio encaminado a su comprensión cabal es requerido y bienvenido.

Segunda parte

Esta sección comienza con un tratamiento de los fundamentos de la programación como si ésta fuera una ciencia formal, dotada de axiomas y reglas de inferencia para construir los "teoremas", *i.e.*, los programas. Se empieza con la explicación del concepto de algoritmo y de su conceptualización matemática en términos de la máquina de Turing, para definir a continuación las

herramientas fundamentales para la construcción de programas, que no serán sino la representación, en el modelo de von Neumann, del concepto de algoritmo ya estudiado. El quinto capítulo concluye con un estudio introductorio al ciclo de vida de un programa, y hace referencia al análisis de sistemas en programación. Es aquí donde se explican los esquemas de *programación estructurada* y *programación en pseudocódigo* y se estudia la diferencia entre las actividades de programación y codificación. Se manejan también otros conceptos relevantes como *estructuras de control, estructuras de datos* y *diseño estructurado.*

En el sexto capítulo se desarrollan los conocimientos anteriores y se proponen técnicas de construcción estructurada de programas. Se precisan los criterios para construir programas completos, y se explica y analiza una metodología para el diseño de programas y sistemas de computación. Para ilustrar lo hasta ahí dicho se incluyen también varios ejemplos terminados de programas codificados en cinco diferentes lenguajes de programación.

Una vez aprendidos y practicados suficientemente los conceptos básicos de la programación, llega el momento de ponerlos en ejecución en una computadora, para lo cual se vuelve necesario utilizar un lenguaje de programación en particular. Se escogieron dos de ellos: uno "antiguo", FORTRAN, y otro reciente, Pascal, para ilustrar todo lo estudiado.

El séptimo capítulo, el último, no tiene como objetivo central enseñar estos dos lenguajes, sino usarlos como vehículo para llevar a buen fin los conocimientos adquiridos, y deberá leerse con este espíritu. No obstante, sin duda, el lector que tenga acceso a una computadora para experimentar podrá, sin demasiado esfuerzo, convertirse en un programador eficiente que aporte su contribución a la gran tarea de construir nuestro camino en la tecnología.

Una de las ideas fundamentales de esta segunda parte la constituye aclarar que es más importante programar que codificar. Esto es, se destaca la necesidad de partir de una idea completa de qué se va a hacer, antes de poner manos a la obra y tropezar con los acostumbrados problemas que se tienen cuando ya es demasiado tarde y se está inmerso en un lenguaje de programación particular, sin haber tenido previamente claro el problema.

En fin, este libro es para lectores y estudiantes dispuestos a afrontar los desafíos que la ciencia y la tecnología presentan, con todos sus peligros y dependencias, pero también con el enorme potencial social que representan.

Plan de estudio

La experiencia académica demuestra que el presente texto es adecuado para un primer curso intensivo de computación en escuelas y facultades de ingeniería o ciencias. (Se ha empleado en la unidad Iztapalapa de la Universidad Autónoma Metropolitana, dentro del programa educativo de las licenciaturas en computación e ingeniería electrónica.) En planes de estudio de menor alcance, se recomienda para dos cursos: el primero, de introducción general —pero rigurosa— a las ciencias de la computación, y el segundo con fines más operativos, tendiente a que los estudiantes adquieran habilidades prácticas en programación de computadoras.

Los estudiantes de disciplinas tradicionalmente consideradas ajenas a las matemáticas y las ciencias exactas, como administración y ciencias sociales, encuentran cada vez con mayor frecuencia que deben entrar en contacto con la computadora, y no como simples usuarios finales, sino como interesados directos. Se les recomienda dedicar tiempo a un estudio serio en la materia, que vaya más allá de las generalidades, aunque la profundidad en el tratamiento de los temas no pueda ser la misma que logran ingenieros o matemáticos.

A continuación se sugieren dos planes de estudio que abarcan desde una sencilla introducción a la computación hasta dos cursos completos y un laboratorio de prácticas de programación con una computadora, y un plan intermedio que considera los conocimientos básicos, sin incluir habilidades prácticas en programación*.

PLAN I ESTUDIO INTRODUCTORIO GENERAL

- -Capítulo 1
- -Introducción del capítulo 2
- -Sección 3.6
- -Resumen del capítulo 4
- -Sección 6.2

PLAN II ESTUDIO INTRODUCTORIO INTERMEDIO

- -Capítulo 1
- -Capítulo 2
- -Capítulo 3
- -Resumen del capítulo 4
- -Capítulo 6
- -Secciones 7.1 y 7.2

* Por motivos prácticos, estas sugerencias son sobre el nuevo contenido de la segunda edición.

PLAN III ESTUDIO COMPLETO: DOS CURSOS Y UN LABORATORIO DE PROGRAMACIÓN

PRIMER CURSO: Capítulos 1 a 4 (PARTE I)
SEGUNDO CURSO: Capítulos 5 a 7
REFERENCIAS EXTRA: Apéndices A y B
LABORATORIO DE PROGRAMACIÓN: Capítulo 8

Agradecimientos

Todo proyecto de alguna consideración tiene siempre antecedentes, que pueden incluso estar lejanos en el tiempo; en este caso quisiera mencionar al Ing. Edmundo Ponce Adame, funcionario de la Universidad de Guadalajara, y a Carlos M. Péretz, quienes me ofrecieron la oportunidad de iniciarme en el campo de la computación, así como al Dr. Enrique Grapa, el Dr. Luis Legarreta y el maestro Micael Cimet.

La primera versión del libro fué escrita en su totalidad con el procesador de palabras de la minicomputadora ONYX, que funciona con el sistema operativo Unix. Esta máquina se encuentra en las instalaciones de los Laboratorios de Investigación e Informática, S. A. (LIISA), del Grupo Micrológica. Deseo agradecer a mis compañeros de trabajo la comprensión y apoyo prestados durante el tiempo que estuve dedicado a terminarlo; en particular, a Alejandro González, quien leyó el original completo. Alfredo Sánchez, José Quiroga, Alfonso Zamarripa y Luis Orozco también aportaron sus valiosos puntos de vista. En la parte de Pascal igualmente recibí la ayuda de Rafael López Barrio y de Víctor Tapia.

Agradezco a la Dra. Sara Poot Herrera el tiempo que dedicó a leer el texto, así como sus sugerencias y correcciones de estilo para hacer más fluida la lectura.

El Dr. Diego Bricio Hernández amablemente leyó el material completo e hizo valiosas sugerencias, algunas de ellas sorprendentes por su originalidad.

De la misma forma, agradezco al Dr. Renato Barrera el tiempo que dedicó a la lectura de la primera versión del libro, así como sus comentarios.

El Dr. Moisés Moshinsky enriqueció sustancialmente el proyecto original con sus comentarios, críticas y aportaciones, que en mucho aprecio.

José de Jesús Muñoz, Gerente Editorial de la División Universidades de McGraw-Hill, me obsequió con su constante apoyo y fe en este proyecto; sin él hubiera sido imposible llevarlo a buen término.

Recuerdo y aprecio también las agradables discusiones, mitad filológicas y mitad filosóficas, que sostuve con el Lic. Joaquín Mejía Gómez, supervisor de traducción y corrección de la editorial.

Los dos dibujos originales que aparecen en el glosario son de mi amigo Ariel Guzik, quien además es un experto en diseño de computadoras.

Agradezco al Sr. Roberto Fuentes C., supervisor de producción de McGraw-Hill, su atención y esfuerzo en la etapa final de este libro.

Por último, agradezco al M. en C. Roberto Romo Ríos su participación en una revisión final de todo el material aquí presentado.

Es importante aclarar, sin embargo, que de antemano asumo toda la responsabilidad por los errores que aún pueda contener el texto, o por la polémica que —espero— pueda levantar en términos del enfoque pedagógico propuesto.

Quien hace lo que puede
hace lo que debe.
Proverbio tradicional francés

Ante la notoria carencia de material didáctico introductorio escrito en español, que sea producto de nuestra experiencia universitaria y no simple traducción de textos extranjeros, se entrega éste, esperando que sirva de compañero inicial en el estudio de esta ciencia tan apasionante como bella.

Guillermo Levine

Primera parte

1

Resumen histórico de la computación

1.1 Antecesores y razón de ser

La computadora representa, de alguna manera, el genio encerrado en la botella, pues es capaz de cumplir los deseos de rapidez y eficiencia en el cálculo y la organización de grandes masas de datos. Es ya común oír que la sociedad moderna depende de las computadoras, y cada vez con mayor frecuencia se escucha que nuestro futuro está ligado al de estas máquinas. Por ello, resulta interesante averiguar de dónde surgieron las computadoras, y más aun enterarse de cómo surgió la idea que las sustenta, porque está claro que ningún invento de alguna importancia surge aislado de una conceptualización previa, que a veces lo antecede por muchos años.

En el caso de las computadoras, aunque aparecen a finales de la década de 1950, las ideas de las que provienen son en realidad tan básicas y primordiales para el ser humano que no deja de ser extraño que muchos estudiantes y usuarios no las conozcan. Además, al ignorarlas se corre el peligro de creer que las computadoras son meros artefactos para calcular, con lo que se pierde el acceso a un gran conjunto de conceptos filosóficos.

Consideraciones filosóficas

Aunque parezca extraño, la filosofía es el punto de partida de este libro sobre computación, y en un somero análisis se explicará el porqué.

Si se piensa en los medios de que los humanos disponemos para conocer el mundo, hay que considerar en primer lugar la percepción que otorgan los sentidos; y al analizar esto con detenimiento se encuentra que, en principio, no percibimos el mundo en forma directa, sino a través del complejo mecanismo de los sentidos, que confiere a nuestra interpretación características

propias. Cuando el ser humano adquiere el manejo del lenguaje, a los pocos años de edad, la situación da un giro radical; entonces, la percepción del mundo es aun menos directa que antes y se convierte, en buena medida, en un conjunto de descripciones acerca de él en términos del lenguaje. Estas descripciones son imágenes mentales estructuradas que todo el tiempo nos dicen qué y cómo es el mundo en un permanente "diálogo interno". Haga el lector un pequeño experimento para ilustrar esto: intente observar el segundero de un reloj durante un minuto anulando totalmente dicho diálogo interno y se dará cuenta de que es casi imposible*.

Se puede entonces postular que uno de los mecanismos mediante los cuales el ser humano conoce el mundo es el de las *descripciones* que de él constantemente hace, y que estas descripciones están construidas mediante el lenguaje.

Una característica primordial de este mecanismo es que podemos transmitir las descripciones a alguien más y esperar que éste las procese de tal forma que sea capaz de comprender que la descripción emitida hace referencia a una misma realidad percibida por ambos. El proceso mediante el cual se logra esta comprensión extrae, mediante una *representación*, el contenido original de la descripción, y va de regreso al punto de partida inicial. De esta manera es como se puede establecer un primer nivel de comunicación acerca del mundo, aunque todavía quedan muchos aspectos pendientes que se estudiarán con más detenimiento en el capítulo 5 de este libro, cuando se analice el problema de la computabilidad, que en principio se dedica a explorar en términos matemáticos los límites del esquema descripción-representación, y que ha obtenido resultados sorprendentes por lo insospechados.

El proceso de la descripción-representación

Una vez que está claro que es posible representar una descripción, y cuando se han tomado medidas para eliminar los errores de comunicación y las posibilidades de ambigüedad, queda establecido un sistema en el que ya no es indispensable que sea un humano el responsable de realizar la representación; y aquí radica el origen de las computadoras, que está de entrada delimitado por el problema de la computabilidad. En efecto, la computadora no es más que el medio mecánico (o electrónico) con el que se representan descripciones libres de ambigüedad y se obtiene un resultado útil. Estas descripciones, por lo general, se formulan en términos de problemas por resolver mediante un método que la máquina se encarga de representar o llevar a la práctica.

Uno de los problemas que siempre ha fascinado al hombre es el relacionado con la actividad de contar y con el concepto de número. Y de ahí que entre las primeras herramientas que inventó está un ingenio mecánico capaz de li-

* Claro que ésta no es la única manera de conocer el mundo. Uno de los objetivos de la meditación, por ejemplo, consiste en adquirir la capacidad de entrar en contacto directo con el mundo, y no a través de este interminable diálogo interno. El arte obedece también, entre otras, a estas motivaciones, pero éste es un campo de conocimiento que rebasa los alcances de este libro; como introducción mínima a algunos de estos temas se recomienda la referencia [RUSB86] citada al final de este capítulo. (En adelante, las referencias que se indiquen en el texto deberán consultarse al final del capítulo donde aparezcan.)

berarlo de la pesada tarea de calcular a mano. Es más, la misma palabra *cálculo* proviene del latín *calculus*, que nombra las pequeñas piedras que se usaban hace miles de años como auxiliares en las cuentas (en una especie de ábaco formado con ranuras en el suelo y operado manualmente por medio de ellas), y que se han encontrado en excavaciones arqueológicas.

Antecesores de la computadora

El ábaco representa la primera calculadora mecánica, aunque no se le puede llamar computadora porque carece de un elemento fundamental: el programa, que no se logrará sino hasta mucho tiempo después.

Otro ingenio mecánico, que tampoco es una computadora, fue la máquina de calcular inventada por Blaise Pascal (1623-1662). Se trata de una serie de engranes en una caja, que proporcionan resultados de operaciones de suma y resta en forma directa —mostrando un número a través de una ventanita— y que por este simple hecho tiene la ventaja de que evita tener que contar, como en el ábaco; además, presenta los resultados en forma más accesible y directa.

La computadora nace, de hecho, alrededor de 1830, con la invención de la *máquina analítica* de Charles Babbage (1791-1871). Este diseño, que nunca se llevó por completo a la práctica, contenía todos los elementos que configuran a una computadora moderna, y que la diferencian de una calculadora.

La máquina analítica estaba dividida funcionalmente en dos grandes partes: una que ordenaba y otra que ejecutaba las órdenes. La que ejecutaba las órdenes era una versión muy ampliada de la máquina de Pascal, mientras que la otra era la parte clave. La innovación consistía en que el usuario podía, cambiando las especificaciones del control, lograr que la misma máquina ejecutara operaciones complejas, diferentes de las que había hecho antes.

Esta verdadera antecesora de las computadoras contaba también con una sección en la que se recibían los datos con los que se iba a trabajar. La máquina seguía las instrucciones dadas por la unidad de control, las cuales indicaban qué hacer con los datos de entrada, para obtener luego los resultados deseados.

La aplicación fundamental para la que el gran inventor inglés desarrolló su máquina era elaborar tablas de funciones matemáticas usuales (logaritmos, tabulaciones trigonométricas, etc.) que requerían mucho esfuerzo manual.

Esta primera computadora "leía" los datos (argumentos) de entrada por medio de las tarjetas perforadas que había inventado el francés Joseph M. Jacquard, y que habían dado nacimiento a la industria de los telares mecánicos durante la Revolución Industrial.

Conceptualmente, el mecanismo era sencillo: evaluar la primera función $f_1(x_1)$; determinar el nuevo argumento de la serie, x_2, y pedir a la máquina que calculara otra vez la misma función con el nuevo dato. Está claro que si la máquina puede calcular $f_1(x_1)$, no le será difícil calcular $f_1(x_2)$, y de la misma manera podrá generar toda la serie de valores $f_1(x_1), f_1(x_2), \ldots, f_1(x_n)$.

De este modo, si se deseaba calcular una segunda función f_2 sobre un argumento x_1, $f_2(x_1)$, había que cambiar las especificaciones de f_1 por las de f_2,

lo que, supuestamente, se lograba alterando la disposición de ciertos elementos mecánicos en la sección de control de la máquina.

No obstante, la máquina analítica nunca se puso en funcionamiento, precisamente, por la dificultad para lograr dichos cambios. Es perfectamente válido, sin embargo, referirse a esta máquina como la primera computadora digital. (Para mayor documentación sobre este tema consúltense las referencias [GOLH72], [PYLZ75] y [SLAR87].)

Fenómenos analógicos y digitales

Se explicará ahora el significado de la palabra *digital*. Los proceos naturales comparten la característica de ser de tipo continuo; es decir, la escala de manifestaciones de un fenómeno cualquiera no tiene singularidades ni puntos muertos, sino que se extiende de manera continua desde la parte inferior a la superior. La temperatura del agua, por ejemplo, puede variar entre cero y cien grados antes de que ésta cambie de estado; lo importante es que en algún momento el agua puede estar en cualquier punto intermedio de la escala, sin más determinantes que la cantidad de calor que reciba. Asimismo, la velocidad del viento puede fluctuar de manera continua entre cero y cuarenta Km/h en un día normal, pudiendo, en cualquier momento, ocupar una posición en esa escala, sin más limitación que las diferencias de presión atmosférica. Esta característica de poder ocupar cualquier punto intermedio en la escala de manifestaciones, como se mencionó, es común a los fenómenos naturales.

Esto es, en la naturaleza los fenómenos no se limitan a unas cuantas posiciones fijas de sus respectivas escalas de manifestación, sino más bien a una variación **continua** entre dos límites, el superior y el inferior. Los fenómenos que se comportan así reciben el nombre de **analógicos**

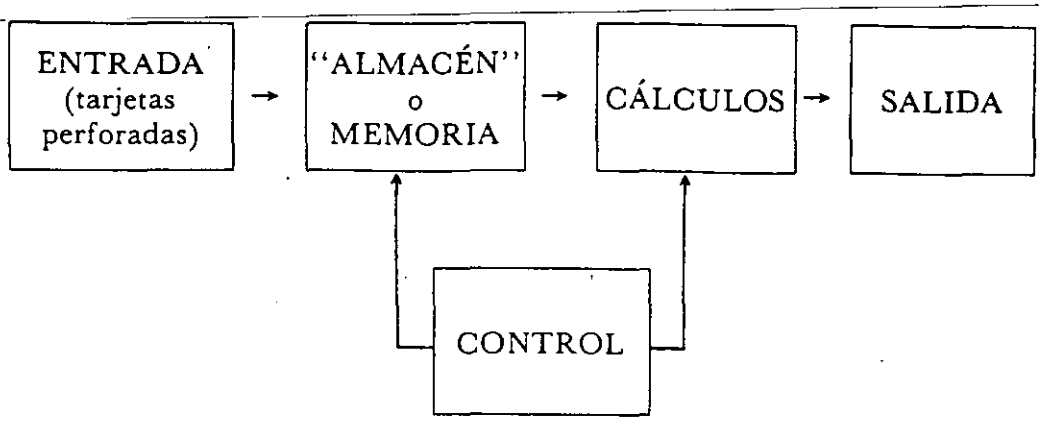
No ocurre lo mismo, sin embargo, con algunos fenómenos creados por el hombre. Piénsese en un automóvil: si se supone de transmisión estándar, entonces se dará el caso que, en algún momento determinado, la caja de velocidades ocupe alguna posición predeterminada (1ª, 2ª, etc.), no pudiendo —más que de manera transitoria— ocupar una posición intermedia. Un automóvil no puede marchar en "primera y tres cuartos"; o lo hace en la primera velocidad o lo hace en la segunda, de manera **discreta** (*i.e.*, discontinua).

Estos fenómenos reciben el nombre de **digitales** (al menos en el contexto de la ingeniería), tal vez porque dan la idea de que se pueden cuantificar con los dedos de la mano.

Así, puede hablarse de computadoras analógicas y computadoras digitales: son computadoras digitales aquellas que manejan la información de manera discreta —en unidades que se llaman bits (*BInary digiTS*, dígitos binarios)—, y son analógicas las que trabajan por medio de funciones continuas —generalmente representación de señales eléctricas.

Hoy día, prácticamente todas las computadoras en uso son digitales, ya que el empleo de las analógicas se ve restringido a aplicaciones muy particulares en la ingeniería o la biología.

A continuación se describe el esquema elemental de la máquina inventada por Charles Babbage, para explicar algunas de las características más importantes de toda computadora digital moderna.



Esquema básico de la máquina analítica

La sección de control se convierte en concepto fundamental, pues es la parte que dirige el procesamiento de acuerdo con un programa previamente introducido en el "almacén" (como llamó Babbage a la memoria) de la máquina. Así, una computadora está formada por una **unidad de entrada**, que recibe tanto la información a procesar como las instrucciones (programa); la **unidad de memoria**, que almacena la información; la **unidad de procesamiento** (aritmética y lógica), que ejecuta los cálculos sobre la información; la **unidad de control**, que dirige a todas las demás unidades, determinando cuándo se debe leer la información, en qué lugares debe almacenarse, cuándo debe funcionar la unidad aritmética, etc.; y una **unidad de salida**, que muestra la información ya procesada, en forma de números o gráficas.

Tiempo después, en 1944, se construyó en la universidad de Harvard, en los Estados Unidos, la computadora IBM Mark I, diseñada por un equipo encabezado por Howard H. Aiken. No obstante, esta máquina no califica para ser considerada la primera computadora electrónica, porque no era de propósito general y su funcionamiento estaba basado en dispositivos electromecánicos, llamados relevadores.

La primera computadora

Luego de casi cien años de Babbage, en 1947, se diseñó la primera computadora electrónica que tenía gran parecido funcional con la máquina analítica —y esto habla de su genio—. Un equipo dirigido por los ingenieros John Mauchly y John Eckert, de la universidad de Pennsylvania, construye una gran máquina electrónica llamada ENIAC (*Electronic Numerical Integrator And Calculator*) que, efectivamente, es la primera computadora digital electrónica de la historia.

Esta máquina era enorme: ocupaba todo un sótano en la universidad, tenía más de 18 000 tubos de vacío, consumía 200 KW de energía eléctrica y requería todo un sistema de aire acondicionado industrial. Pero era capaz de efectuar alrededor de cinco mil operaciones aritméticas en un segundo, dejando para siempre atrás las limitaciones humanas de velocidad y precisión, e inaugurando una nueva etapa en las capacidades de procesamiento de datos.

ENIAC

El proyecto, auspiciado por el Departamento de Defensa de Estados Unidos, culminó dos años después, cuando se integró a ese equipo el ingeniero y matemático húngaro naturalizado norteamericano, John von Neumann (1903-1957). Las ideas de von Neumann resultaron tan fundamentales para su desarrollo posterior, que es considerado el padre de las computadoras.

John von Neumann

La computadora diseñada por este nuevo equipo se llamó EDVAC (*Electronic Discrete Variable Automatic Computer*); tenía cerca de cuatro mil bulbos y usaba un tipo de memoria basado en tubos llenos de mercurio por donde circulaban señales eléctricas sujetas a retardos (para mayor información sobre estos temas véase, por ejemplo, [LUKH79] y [RALA76]).

La nueva idea fundamental resulta muy sencilla: permitir que en la memoria coexistan datos con instrucciones, para que entonces la computadora pueda ser programada de manera "suave"*, y no por medio de alambres que eléctricamente interconectaban varias secciones del control, como en la ENIAC.

Esta idea, que incluso obliga a una completa revisión de la arquitectura de las computadoras, recibe desde entonces el nombre de **modelo de von Neumann**. Alrededor de este concepto gira toda la evolución posterior de la industria y la ciencia de la computación, por lo que se le dedicará un capítulo aparte.

De 1947 a la fecha las cosas han avanzado muy rápido, más que cualquier otro proceso en la historia de la ciencia y la tecnología; a tal grado que en la actualidad hay computadoras mucho más poderosas que la ENIAC y que no ocupan sino un circuito de silicio tan pequeño que es casi invisible

Como contraste inicial, antes de proceder a describir lo que ha sucedido en los cuarenta años desde la invención de la computadora, he aquí una comparación entre dos máquinas, la ENIAC y uno de los primeros microprocesadores (ya obsoleto):

	ENIAC	**Intel 8080**
Año	1947	1973
Componentes electrónicos	18 000 bulbos	Un circuito integrado con más de 100 000 transistores
Tamaño	Decenas de m^2	Menos de 1 cm^2
Requerimientos de potencia	200 kilowatts	Pocos miliwatts
Velocidad	5 000 sumas/s	150 000 sumas/s
Costo	Varios millones de dólares	Cincuenta dólares en 1974

* Tal vez el uso de la palabra *suave* pueda parecer extraño, pero deja de serlo cuando se contrasta con el hecho de que en las computadoras anteriores a von Neumann (y en las actuales calculadoras), las operaciones que se pueden efectuar están "alambradas" y predefinidas, por lo que el usuario no puede cambiarlas. Esta flexibilidad es lo que define a la programación y al término *software* empleado tanto en inglés como en español.

A lo largo de este libro se empleará la palabra *software* porque aún no existe un término equivalente en español —cosa nada rara, puesto que el concepto es de origen anglosajón, y reciente.

Es prácticamente imposible encontrar otro ejemplo de un avance de esta naturaleza en la historia de la tecnología, y esto hace aun más interesante el estudio de lo que ha pasado desde los años iniciales hasta la fecha.

1.2 Generaciones de computadoras

El desarrollo de las computadoras suele dividirse en generaciones. El criterio para determinar cuándo se da el cambio de una generación a otra no está claramente definido, pero resulta aparente que deben cumplirse al menos dos requisitos estructurales:

Criterios para la división en generaciones

A) forma en que están construidas: que haya tenido cambios sustanciales,
B) forma en que el ser humano se comunica con ellas: que haya experimentado progresos importantes.

En lo que respecta al primer requisito, los cambios han sido drásticos en el corto lapso que tienen de existencia las computadoras (desde los tubos de vacío hasta los circuitos microelectrónicos), mientras que el avance en relación con el segundo requisito ha sido más cauteloso. A falta de una definición formal de la frontera entre generaciones, ha surgido una confusión cuando se intenta determinar cuál es la generación actual. Desde un punto de vista estricto (como el que se propone aquí), aún estamos en la tercera generación de computadoras (o en lo que podría llamarse la segunda parte de la tercera generación), porque sólo ha habido adelantos significativos en el punto A), pues en lo relativo al punto B), entre las actuales computadoras y las de hace diez años no hay diferencia sustancial alguna; la comunicación entre el usuario y la máquina sólo se ha vuelto más cómoda y conveniente. Así pues, no está claro si ya estamos en la cuarta generación o si aún no se cumplen los requisitos para el cambio. Sin embargo, la suposición general (avalada fuertemente por los fabricantes de equipo) es que estamos de lleno en la cuarta generación, desde el advenimiento de los microprocesadores, y como ésta resulta ser la opinión más popular (aunque no la más correcta desde nuestro punto de vista), es la que se aplicará en adelante.

Primera generación

Los comienzos de la industria de la computación se caracterizan por un gran desconocimiento de las capacidades y alcances de las computadoras. Así, por ejemplo, según un estudio de la época, serían necesarias alrededor de veinte computadoras para saturar la capacidad del mercado de los Estados Unidos en el campo del procesamiento de datos.

Esta primera etapa abarcó la década de 1950 y se conoce como la primera generación de computadoras. Las máquinas de esta generación cumplen los requisitos antes mencionados de la siguiente manera:

A) por medio de circuitos de tubos de vacío,
B) mediante la programación en lenguaje de máquina (lenguaje binario).

Estas máquinas son grandes y costosas (del orden de decenas o cientos de miles de dólares).

En 1951 aparece la primera computadora comercial, es decir, fabricada con el objetivo de ser vendida en el mercado: la UNIVAC I (*UNIVersAl Computer*). Esta máquina, que disponía de mil palabras de memoria central y podía leer cintas magnéticas, se utilizó para procesar los datos del censo de 1950 en los Estados Unidos. Estos eran los años de la posguerra y la nueva invención aún no presagiaba su gigantesco potencial en la competencia económica internacional, que no llegaría sino hasta una década más tarde.

En las dos primeras generaciones, las unidades de entrada estaban por completo dominadas por las tarjetas perforadas, retomadas a principios de siglo por Herman Hollerith (1860-1929), quien además fundó una compañía que con el paso de los años se conocería como IBM (International Business Machines). En las máquinas de la tercera generación ya se emplean métodos interactivos de comunicación, por medio de pantallas especiales de entrada/salida.

A la UNIVAC I siguió una máquina desarrollada por la compañía IBM, que apenas incursionaba en ese campo; es la IBM 701 (de la que se entregaron 18 unidades entre 1953 y 1956), que inaugura la larga serie por venir.

Posteriormente, la compañía Remington Rand produjo el modelo 1103, que competía con la 701 en el campo científico, y la IBM fabricó la 702, que no duró mucho en el mercado debido a problemas con la memoria.

La más exitosa de las computadoras de la primera generación fue el modelo 650 de IBM, de la que se produjeron varios cientos. Esta máquina usaba un esquema de memoria secundaria llamado tambor magnético, antecesor de los discos que actualmente se emplean.

La competencia contestó con los modelos UNIVAC 80 y 90, que pueden situarse ya en los inicios de la segunda generación. También de esta época son los modelos IBM 704 y 709, Burroughs 220 y UNIVAC 1105.

Segunda generación

Se acercaba la década de 1960 y las computadoras seguían en constante evolución, reduciendo de tamaño y aumentando sus capacidades de procesamiento. Al mismo tiempo se iba definiendo con mayor claridad toda una

nueva ciencia: la de comunicarse con las computadoras, que recibirá el nombre de programación de sistemas.

En esta etapa puede hablarse ya de la segunda generación de computadoras, que se caracteriza por los siguientes aspectos primordiales:

A) están construidas con circuitos de transistores;
B) se programan en nuevos lenguajes llamados lenguajes de alto nivel.

En general, las computadoras de la segunda generación son de tamaño más reducido y de costo menor que las anteriores.

En la segunda generación hubo mucha competencia y muchas compañías nuevas, y se contaba con máquinas bastante avanzadas para su época, como la serie 5000 de Burroughs y la máquina ATLAS, de la universidad de Manchester. Cabe decir que esta última incorporaba —con varios años de anticipación— técnicas de manejo de memoria virtual, que se estudiarán más adelante.

Entre los primeros modelos se puede mencionar la Philco 212 (esta compañía se retiró del mercado de computadoras en 1964) y la UNIVAC M460. Una empresa recién formada, Control Data Corporation, produjo la CDC 1604, seguida por la serie 3000. Estas máquinas comenzaron a imponerse en el mercado de las grandes computadoras.

IBM mejoró la 709 y produjo la 7090 (luego ampliada a la 7094), que ganó el mercado durante la primera parte de la segunda generación. UNIVAC continuó con el modelo 1107, mientras que NCR (National Cash Register) empezó a producir máquinas más pequeñas, para proceso de datos de tipo comercial, como la NCR 315.

RCA (Radio Corporation of America) introdujo el modelo 501, que manejaba el lenguaje COBOL, para proceso administrativo y comercial. Más tarde introdujo el modelo RCA 601.

La segunda generación no duró mucho, sólo unos cinco años, y debe ser considerada como una transición entre las recién inventadas máquinas electrónicas, que nadie sabía con precisión para qué podrían ser útiles, y el actual concepto de computadora, sin el cual el funcionamiento de las modernas sociedades industriales sería difícil de concebir.

Tercera generación

Con la aparición de nuevas y mejores maneras de comunicarse con las computadoras, junto con los progresos en la electrónica, surge la que se conoce como tercera generación de computadoras, a mediados de la década de 1960. Se puede decir que se inaugura con la presentación, en abril de 1964, de la serie 360 de IBM.

Las características estructurales de la tercera generación consisten en:

A) su fabricación electrónica está basada en circuitos integrados* (agrupamiento de circuitos de transistores grabados en pequeñísimas placas de silicio);
B) su manejo es por medio de los lenguajes de control de los sistemas operativos (que se estudiarán más adelante).

Las computadoras de la serie IBM 360 (modelos 20, 22, 30, 40, 50, 65, 67, 75, 85, 90, 195) manejan técnicas especiales de utilización del procesador, unidades de cinta magnética de nueve canales, paquetes de discos magnéticos y otras características que ahora son estándares. No todos estos modelos empleaban esas técnicas, sino que estaban divididos por aplicaciones.

El sistema operativo de la serie 360, llamado simplemente OS (en varias configuraciones), incluía un conjunto de técnicas de manejo de memoria y del procesador que pronto se convirtieron en estándares.

Esta serie alcanzó un éxito enorme, a tal grado que la gente en general, el ciudadano común y corriente, pronto llegó a identificar el concepto de computadora con el nombre IBM. Sin embargo, sus máquinas no fueron las únicas, ni necesariamente las mejores. También en 1964, CDC introdujo la serie 6000, con la máquina modelo 6600, que durante varios años fue considerada como la más rápida.

Esta fue una época de pleno desarrollo acelerado y de competencia por los mercados internacionales, ya que la industria de la computación había crecido hasta alcanzar proporciones insospechadas. Es curioso reflexionar en que estos años coinciden con el "retroceso" racional y la "vuelta a los orígenes" planteados por una juventud rebelde y sospechosa de la supuesta invasión tecnológica.

Al inicio de la década de 1970, IBM produce la serie 370 (modelos 115, 125, 135, 145, 158, 168), que representa una mejora (aunque no radical) a la serie 360. UNIVAC compite con los modelos 1108 y 1110, máquinas de gran escala; mientras que CDC inaugura su serie 7000 con el modelo 7600, mejorado después para producir la serie Cyber. Estas computadoras son tan potentes y veloces que se convierten ya en un asunto de estado y de seguridad nacional para el país que las produce, y se cuida, ya en los más altos niveles gubernamentales, su exportación y comercialización internacional.

A finales de esa década, IBM introduce las nuevas versiones de la serie 370 con los modelos 3031, 3033 y 4341, en tanto que Burroughs participa con las computadoras de la serie 6000, (modelos 6500, 6700) de avanzado diseño, luego reemplazadas por la serie 7000. La compañía Honeywell participa con las computadoras de la línea DPS, en varios modelos.

* Estos microcircuitos reciben el nombre de circuitos integrados, y son conocidos también por su nombre popular en inglés, *chip*. Su origen se remonta a 1958-59, cuando la idea de obtener e interconectar capacitores y transistores a partir de un pequeño bloque de silicio se les ocurre, en forma independiente, al Dr. Robert Noyce, de la recién creada compañía Fairchild Semiconductors, y al ingeniero Jack Kilby, de Texas Instruments.

En Japón la compañía Fujitsu produce computadoras poderosas, que van desde máquinas relativamente pequeñas hasta verdaderos gigantes (de la serie FACOM), comparables sólo con los más grandes sistemas de CDC o IBM.

Las grandes computadoras reciben en inglés el nombre de *mainframes*, que significa, precisamente, gran sistema.

Entre las máquinas de la tercera generación hay algunas dedicadas a procesos especiales, que manejan cientos de millones de números en representación decimal y requieren diseños específicos para ser resueltos (para más información consúltese el artículo [LEVR82]).

Minicomputadoras

A mediados de la década de 1970 (en plena tercera generación) surge un gran mercado para computadoras de tamaño mediano, o **minicomputadoras**, que no son tan costosas como las grandes máquinas, pero que ya disponen de una gran capacidad de proceso. En un principio, el mercado de estas nuevas máquinas estuvo dominado por la serie PDP-8 de DEC (Digital Equipment Corporation), actualmente en desuso.

Otras minicomputadoras populares son la serie PDP-11 de DEC, reemplazada por las nuevas máquinas VAX (*Virtual Address eXtended*) de la misma compañía, los modelos Nova y Eclipse de Data General, las series 3000 y 9000 de Hewlett-Packard, en varias configuraciones, y el modelo 34 de IBM, que luego fue reemplazado por los modelos 36 y 38.

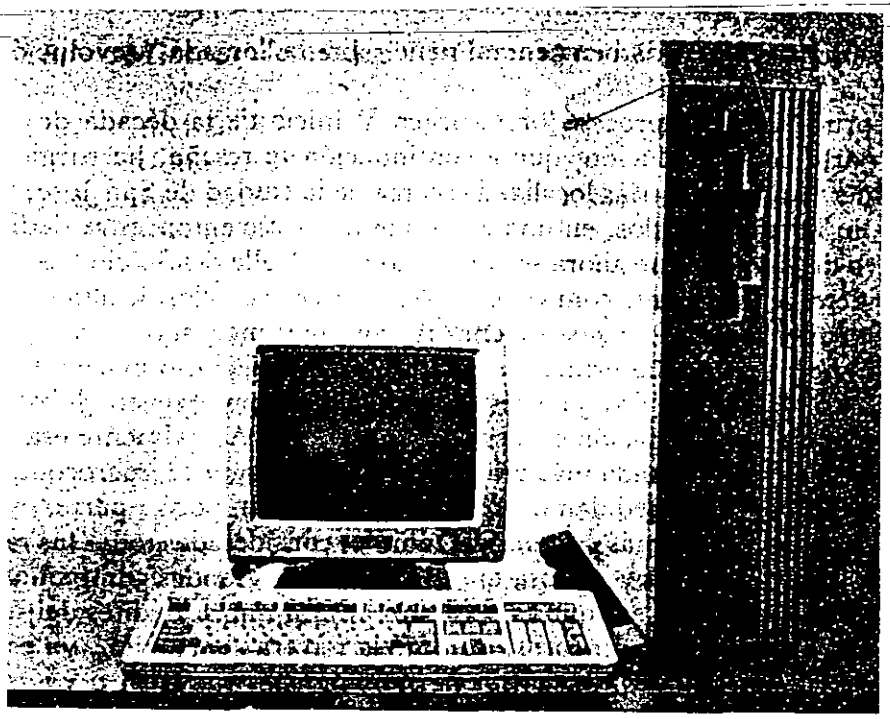
Dentro de esta categoría caben también las máquinas Wang y Honeywell-Bull, en diversas configuraciones, así como computadoras Prime, ICL (International Computers Limited, inglesa), Siemens (alemana), etcétera.

En la Unión Soviética son de amplio uso las computadoras de la serie SU (Sistema Unificado, Ryad), que también han pasado por varias generaciones. La primera de estas máquinas era, en cuanto a la arquitectura, una copia de la serie 360 de IBM, con los modelos ES 1020 a 1060. A fines de la década de 1970 surgió la serie Ryad-2, cuya arquitectura sigue a la de la serie 370.

Asimismo, los países socialistas han desarrollado una serie de computadoras dedicadas al control industrial, además de las máquinas de la serie Minsk y BESM (véase el artículo [DAVC78] para una descripción).

Cuarta generación

El adelanto de la microelectrónica prosigue a una velocidad impresionante, y ya por el año de 1972 surge en el mercado una nueva familia de circuitos integrados de alta densidad, que reciben el nombre de **microprocesadores**. Las **microcomputadoras** que se diseñan con base en estos circuitos son extremadamente pequeñas y baratas, por lo que su uso se extiende al mercado de



Cortesía del Grupo Micrológica

consumo industrial. Hoy día hay microprocesadores en muchos aparatos de uso común, como relojes, televisores, hornos, juguetes, etc.; y naturalmente, en toda una nueva generación de máquinas, aunque sólo en lo que respecta al equipo físico (requisito A antes mencionado), puesto que en el otro aspecto (requisito B para determinar el cambio de una generación a otra) no ha habido progresos de esta magnitud, aunque los cambios habidos tampoco son despreciables.

Sin embargo, como se señaló antes, lo usual es suponer que nos encontramos en la cuarta generación, e incluso hay quien comienza a hablar de la quinta, por lo que más adelante también se menciona.

La industria de la microelectrónica se ha colocado entre las más grandes ramas económicas de la sociedad industrial, y se calcula que para el año 2000 será la segunda en nivel mundial, superada tan sólo por la agricultura. Para dar una idea, en 1986 produjo ventas por 300 000 millones de dólares.

1.3 Microcomputadoras y computadoras personales

Es conveniente hacer un recuento de los diferentes microprocesadores que han aparecido en los últimos quince años, porque han dado lugar al nacimiento de toda una industria (de las computadoras personales) que ha adquirido proporciones enormes, y que ha influido poderosamente sobre la

percepción que la sociedad en general tiene sobre la llamada "revolución informática".

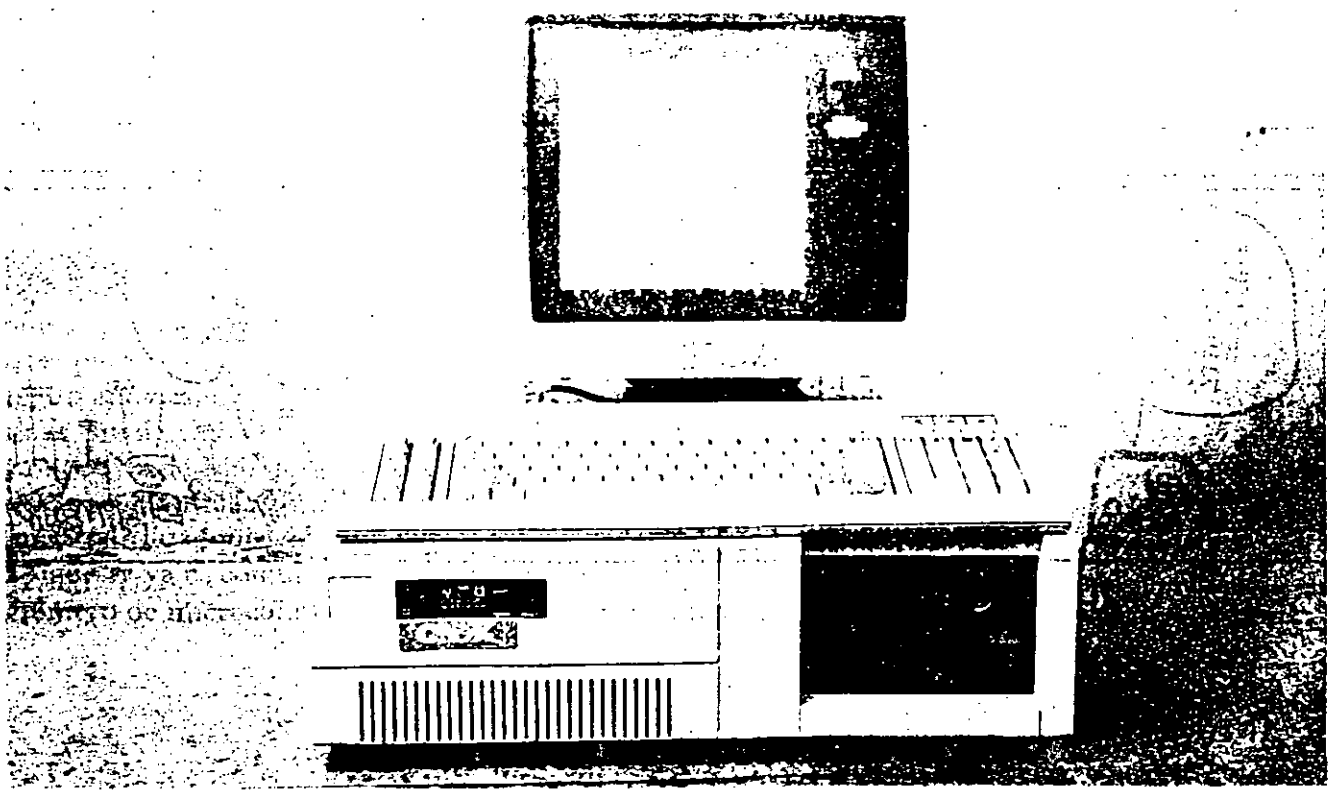
Los primeros microprocesadores surgen al inicio de la década de 1970. Como parte de su evolución, que a continuación se reseña, ha surgido un conglomerado de industrias localizadas cerca de la ciudad de San José, California, en Estados Unidos, en una zona que hasta ese entonces se dedicaba al cultivo de la vid, y que ahora se conoce como el Valle del Silicio. Esa zona se considera generalmente como el corazón de los desarrollos de alta tecnología en microelectrónica, y sus orígenes fueron muy modestos, como lo evidencia el hecho de que la primera microcomputadora de uso masivo fue inventada en 1976 por dos jóvenes (Steve Wozniak y Steven Jobs) que trabajaban en una habitación desocupada de su casa. Actualmente esa compañía (Apple) es la segunda más grande de la industria de la microcomputación en el mundo, antecedida tan sólo por IBM; y ésta, por su parte, es una de las cinco compañías más grandes del mundo, consideradas todas las ramas de la industria, y es más poderosa que casi todas las grandes compañías petroleras y automotrices, aunque es necesario aclarar que aun antes del surgimiento de las microcomputadoras, durante la tercera generación, ya estaba entre los primeros lugares. Las cifras de 1986 indican que IBM tuvo ventas totales por más de 50 000 millones de dólares, y su planta está integrada por más de 400 000 empleados.

En la lista que sigue se mencionan sólo los microprocesadores más conocidos, y las fechas corresponden a cuando ya eran un producto accesible en el mercado y no únicamente muestras de laboratorio.

Año	Microprocesadores	Comentarios
1969	Intel 4004	No fue un producto comercial.
1971	Intel 8008	Primer microprocesador de 8 bits.
1973	Intel 8080	Surge la industria de las microcomputadoras.
1974	Motorola 6800	
1975	Zilog Z80	Aparece el sistema operativo CP/M.
		Aparece la Apple.
	Intel 8085	Inicia el auge de la microcomputación.
1976	Intel 8085	Aparece la IBM 5100.
	Mostek 6502	
1978	Motorola 6809	
1981	Zilog Z8000	Microprocesadores de 16 bits.
		Surgen las microcomputadoras con el sistema operativo UNIX.
	Intel 8088	Aparece la computadora IBM-PC, con el sistema operativo MS-DOS.
		Surge una industria paralela de computadoras personales.

Año	Microprocesadores	Comentarios (*Continuación*)
1982	Intel 8086	
	Motorola 68000	
1984	Intel 80186	Aparecen las computadoras PC-AT.
	Intel 80286	
	Motorola 68010	
	National 32000	
1986	National 32032	Microprocesadores de 32 bits.
	ATT WE32100	
	NEC V70	
	Zilog Z280	
	Intel 80386	
		IBM anuncia la serie PS/2, cuyo sistema operativo, OS/2, comenzará a operar en 1988.
1987	Motorola 68020	
	Fairchild Clipper	
1988	Motorola 68030	
	Motorola 78000	

Las microcomputadoras basadas en estos (y otros) microprocesadores son de marcas tan diversas como Altos, Apple, ATT, Commodore, Compaq, Epson, Hewlett-Packard, IBM, NCR, NEC, Olivetti, Onyx, Tandy y Toshiba, entre muchísimas otras.



En 1981 se vendieron 800 000 computadoras personales, cifra que al año siguiente subió a 1 400 000. Entre 1984 y 1987 se han vendido en todo el mundo más de 60 millones de computadoras personales, por lo que no cabe duda de que su impacto y penetración han sido enormes. En 1984, por primera vez sucedió que las ventas globales de computadoras personales sobrepasaron a las reproducidas con los grandes sistemas. Un curioso estudio indica que todas las computadoras personales y los microprocesadores juntos alcanzarían a procesar la astronómica cifra de 20 billones de instrucciones por segundo, mientras que la potencia sumada de todas las grandes computadoras existentes alcanzaría "tan sólo" para 145 mil millones de instrucciones por segundo.

Sobre todo desde el surgimiento de las computadoras personales, en 1981, el software y los sistemas que con ellas se manejan han tenido un considerable avance, porque han vuelto mucho más interactiva la comunicación con el usuario. Existe toda una familia de aplicaciones (procesadores de palabras, hojas electrónicas de cálculo, paquetes gráficos, etc.) que, aunque ya existían en la tercera generación, han alterado de manera significativa el potencial y la flexibilidad de uso de los sistemas, por lo cual se han ganado un lugar dentro de la programación de sistemas. (En la sección 4.7 se hace una somera descripción de ellas.) La industria del software de las computadoras personales también ha crecido con gran rapidez, y tan sólo en 1986 produjo sistemas por un valor de 3 300 millones de dólares en los Estados Unidos. Parte de esta explosión en la industria del software se debe a individuos como Gary Kildall y William Gates (creadores de CP/M y de los productos Microsoft, respectivamente), jóvenes que se dedicaron durante años a la creación de sistemas operativos y métodos para lograr una utilización sencilla de las microcomputadoras.

Ya resultan comunes las microcomputadoras de uso personal, lo suficientemente baratas y accesibles para ser empleadas por pequeñas organizaciones y negocios, donde se destinan a tareas como control de nómina, contabilidad

EL CUARTO REICH ■ Palomo

Reproducida con autorización del autor

e-inventarios. También se ha extendido su uso en aplicaciones creativas en computación y como pasatiempo. Si bien se habla de una revolución social causada por las computadoras, refiriéndose a que su uso —sobre todo el de las personales— abarca muchos estratos sociales, esto no es totalmente cierto ni siquiera en los países desarrollados. No obstante, sí se ha abierto un nuevo campo de estudio para la sociología, que debe dilucidar el impacto social de esta tecnología.

Aunque no es del todo correcto suponer que estas nuevas máquinas revolucionarán la manera en que pensamos, sí es evidente su enorme potencial para automatizar buena parte de las tareas administrativas usuales, por un lado, y para agilizar el aprendizaje y el uso de la computación, por el otro.

No todo son microcomputadoras, por supuesto; las minicomputadoras y los grandes sistemas continúan en desarrollo. Ya es un hecho que máquinas pequeñas rebasan por mucho la capacidad de los grandes sistemas que hace diez o quince años requerían instalaciones costosas y especiales, pero sería equivocado suponer que las grandes computadoras han desaparecido; muy por el contrario, su presencia es ya ineludible en prácticamente todas las esferas de control gubernamental, militar y de la gran industria. Las enormes computadoras de las series CDC, CRAY, Hitachi o IBM, por ejemplo, son capaces de atender a varios cientos de usuarios simultáneamente, y su velocidad de procesamiento sobrepasa los cientos de millones de operaciones por segundo.

Hoy día hay multitud de fabricantes de equipos de todos tamaños. Además de los ya mencionados, se puede considerar como importante la fusión que se dio en 1986 entre las compañías Sperry-Univac y Burroughs, para formar Unisys. También son de consideración los fabricantes de las computadoras de la serie Amdhal (que son competencia directa de los grandes equipos IBM), y los diseñadores de las computadoras Tandem y las estaciones de graficación y diseño Sun. Cabe mencionar que la industria de la computación es muy dinámica y competitiva, por lo cual la movilidad de las compañías también es muy grande.

El desarrollo computacional en la sociedad

El motor de este dinámico mundo de las computadoras existe sólo en algunos países desarrollados, y el resto de la comunidad internacional se ve limitada a desempeñar el papel de consumidor de la tecnología que se produce a un enorme costo en las sociedades industriales. Además, la comercialización de las computadoras no siempre está en función de los intereses de los países que las adquieren, y hay más de un caso en el que se han vendido miles de computadoras sin tener claro previamente un plan racional de utilización. Tómese por ejemplo el caso de una compra de 300 millones de dólares en partes para microcomputadora que hizo China en 1984, y que dio lugar a que, un año después, estuvieran almacenadas en bodegas varios miles de máquinas, sin destino ni utilización claros. También ocurre que, en países como Zaire, Costa de Marfil y Kuwait, entre otros, el equipo instalado de computadoras se subutiliza en más de un 50%, según estudios internacionales. La carrera por la adquisición —muchas veces indiscriminada— de computadoras sigue, y ya es común que en países como Senegal y Gabón, por ejemplo, el número de microcomputadoras se incremente de menos de 700 y 400, res-

pectivamente, en 1984, a casi el triple en sólo un año. Para las grandes firmas exportadoras, los países del tercer mundo representaron, durante 1985, tan sólo el 6% de su producción mundial, por lo que está claro que lo que estas naciones requieren no necesariamente es del interés prioritario de las grandes corporaciones. La dependencia tecnológica no sólo no ha desaparecido, sino que va en aumento.

Esfuerzos en países del tercer mundo

Hay algunos esfuerzos interesantes en países del tercer mundo por revertir esto. El caso más interesante lo presenta Brasil, que en 1984 promulgó un plan nacional de protección a la industria propia de computación, de ocho años de duración, con resultados relativamente halagadores a la fecha, y que ha logrado que este país sea casi autosuficiente en la producción de mini y microcomputadoras, aunque parte del diseño original y muchos de los microcircuitos aún se importan. Otros países que han tomado medidas para proteger su industria de computación han sido India y Corea del Sur, aunque en esos casos la finalidad ha sido convertirse en exportadores de microcomputadoras de precio competitivo, aunque no de diseño avanzado. México ha dado algunos pasos en ambas direcciones (crear tecnología propia y producir para exportación), aunque no con la firmeza y el apoyo nacional requeridos.

La competencia internacional es enorme y desproporcionada, y no es casual que también en este campo las relaciones entre las naciones estén marcadas no por un afán de cooperación, sino por el de optimización de los beneficios económicos. Para dar una idea de la desproporción, compárense estas cifras, correspondientes a la producción de equipos para procesamiento de datos en 1985. Las cantidades están dadas en millones de dólares.

País	Ventas
Brasil	463
Corea del Sur	235
Estados Unidos	47 069
Hong Kong	796
India	40
Japón	20 909
Singapur	588
Taiwán	377

En la Unión Soviética las cosas no han avanzado con tanta rapidez. Se estima que en la actualidad la industria de la microelectrónica tiene entre cuatro y seis años de atraso tecnológico comparada con la de los Estados Unidos y Japón. Son de uso cada vez más extendido las mini y microcomputadoras de la familia Elektronika S5, y se ha despertado gran interés por las computadoras personales Agat e Iskra. Algunos países del área de influencia soviética (Hungría y Yugoslavia sobre todo) se han convertido en exportadores de unidades periféricas de disco y cinta magnética, incluso a Occidente.

Quinta generación

En vista de la acelerada marcha de la microelectrónica, la sociedad industrial se ha dado a la tarea de poner también a esa altura el desarrollo de software y los sistemas con los que se manejan las computadoras. Ha surgido un interesante fenómeno de competencia internacional por el dominio del gigantesco mercado de la computación, en el que se perfilan dos líderes que, sin embargo, no han podido aún alcanzar el nivel que se desea: la capacidad de comunicarse con la computadora mediante el lenguaje natural y no a través de códigos o lenguajes de control especializados.

Japón lanzó en 1983 el llamado "programa de la quinta generación de computadoras", con los objetivos explícitos de producir máquinas con innovaciones reales en los dos criterios mencionados. Y en Estados Unidos ya está en actividad un programa de desarrollo que persigue objetivos semejantes, que pueden resumirse de la siguiente manera:

A) procesamiento en paralelo mediante arquitecturas y diseños especiales y circuitos de gran velocidad;
B) manejo de lenguaje natural y sistemas de inteligencia artificial.

El futuro previsible de la computación es muy interesante, y se puede esperar que esta ciencia siga siendo objeto de atención prioritaria de gobiernos y de la sociedad en conjunto.

Termina esta sección recordando al lector que todas estas computadoras (que no son las veinte que predecía el estudio de 1951) están basadas en la idea de von Neumann, por lo que se procederá a analizarla en el siguiente capítulo.

1.4 Anexo: tecnología de microcomputadoras*

En virtud de la facilidad de acceso a las computadoras personales y a la enorme cantidad de recursos, unidades periféricas, procesadores especializados, interfaces de entrada y salida, interfaces gráficas y sistemas que existen, se vuelve necesario hacer un resumen de todo este universo; por ello en este espacio se hace una breve descripción de la nomenclatura en uso, clasificada de acuerdo con la función que desempeña cada dispositivo utilizado en una microcomputadora.

El nacimiento de las microcomputadoras tuvo lugar en los Estados Unidos, como se menciona en el capítulo, a partir de la comercialización de los

* Debo la información fuente de este anexo a mi amigo Alfredo Sánchez A., presidente de Onyx Technologies, California, Estados Unidos.

primeros microprocesadores de 8 bits (Intel 8008, 8080) a comienzos de la década de 1970. Las primeras de estas máquinas se conocían sencillamente como microcomputadoras, y comenzaron a tener aceptación, primero exclusivamente en el mercado de los técnicos e ingenieros que deseaban (o podían) experimentar con esta nueva tecnología, que ponía al alcance de su mano un tipo de equipo que antes les era completamente inaccesible. La empresa IMSAI vendía una microcomputadora en forma de partes que el entusiasta experimentador armaba y probaba. Pronto otras compañías entraron a la competencia, y algunas comenzaron a ofrecer equipos ya armados, más fáciles de utilizar, aunque seguía siendo absolutamente necesario un conocimiento especializado sobre la materia (o al menos la disposición de dedicar muchas horas al día a obtenerlo). Durante la década de 1970 aparecieron microcomputadoras de marcas como Altos, Apple, Cromemco, Heathkit, IMS, Ohio y Zenith, y pronto se impusieron dos tendencias: la de los sistemas Apple, que empleaban su propia tecnología (basada en el microprocesador Mostek 6502) y un sistema operativo particular, y la de casi todas las demás microcomputadoras, que empleaban el sistema operativo CP/M (*Control Program/Monitor*), basadas en los procesadores Intel 8080 y 8085, y luego en el Zilog Z80. Durante varios años la situación se mantuvo con un ritmo de crecimiento rápido, pero frenado aún por la dificultad de empleo de estos dispositivos.

Cuando el mercado ya había adquirido proporciones importantes, y una vez que la nueva tecnología estaba ya aceptada, comenzó la verdadera explosión comercial masiva, con la introducción, en 1981, de la *Personal Computer* de IBM. Esta máquina (basada en el microprocesador Intel 8088) tenía características interesantes, que hacían mucho más amplio su campo de aplicaciones, sobre todo porque tenía un nuevo sistema operativo estandarizado (MS-DOS, *Microsoft Disk Operating System*) y una capacidad mejorada de graficación, lo que la hacía más atractiva y relativamente más fácil de usar.

En la actualidad existen varios centenares de fabricantes de computadoras personales (o de partes), mayoritariamente distribuidos en los Estados Unidos, Japón, Taiwán, Corea del Sur y Europa, y juntos configuran una enorme industria que ha colocado decenas de millones de microcomputadoras en el mercado durante la última década, lo que ha dado lugar también al nacimiento de tecnologías nuevas, tanto de diseño como de producción y empaque; con ello también se ha dado lugar al nacimiento de una terminología técnica nueva. El mercado de este tipo de computadoras ha estado creciendo a un ritmo del 10% anual en los Estados Unidos, (aunque algunos años ha sido menor) por lo que se espera que para 1991 las operaciones en este mercado asciendan a 24 mil millones de dólares, según cifras del Departamento de Comercio de ese país. Para tener un punto de referencia, considérese que el Producto Interno Bruto de México en 1986 fue de aproximadamente 120 mil millones de dólares.

A la cabeza de esta industria están, prácticamente, sólo dos compañías: IBM y Apple; juntas manejan cerca del 50% del mercado mundial de las computadoras personales, aunque sus modelos son mutuamente incompatibles. Como se dijo, IBM lanzó en 1981 la computadora personal PC y deter-

minó la dirección a seguir durante media década, hasta que en 1987 descontinuó la línea e inauguró una nueva, llamada PS/2 (*Personal Systems*), con varios modelos basados en los procesadores Intel 8086, 80286 y 80386. Apple, por su parte, mantiene vigentes sus familias Apple y Macintosh (basada en el procesador Motorola 68000), en varias versiones. Asimismo, muchas compañías han aprovechado el hecho de que desde el inicio IBM hizo pública la arquitectura (o sea, el diseño global) de su computadora PC para producir una gran cantidad de computadoras parecidas, tanto en funcionamiento como en apariencia, de tal forma que es común la existencia de modelos (a veces más poderosos y económicos) similares, conocidos en inglés como *clones*: dobles. En vista de la gran competencia que estas máquinas "apócrifas" hicieron a las de IBM, esta compañía decidió mantener privado el diseño y especificaciones de la nueva serie PS/2, por lo que no han surgido aún competidores directos, aunque se espera que muy pronto aparezcan, dado el gran dinamismo de esta industria.

Sin embargo, a partir de que IBM abandonó la línea PC, este segmento del mercado ha seguido creciendo, gracias a la introducción de modelos cada vez más poderosos, basados en microprocesadores de 32 bits, y debido también a la existencia de varias decenas de millones de computadoras tipo PC que siguen requiriendo sistemas, unidades periféricas y apoyo comercial.

Originalmente, la computadora personal surgió para atender las necesidades de los usuarios individuales, en las ramas de procesamiento de palabras, hojas electrónicas de cálculo, gráficas y aplicaciones varias, que en general no están directamente relacionadas con otras ramas de la operación de una oficina o industria. Para dar un ejemplo, en una fábrica con varios centenares de empleados no es factible dedicar una computadora personal a las labores de mantenimiento de los inventarios, porque la capacidad de la pequeña máquina es insuficiente, y porque para atender esta labor se requieren varios puntos de control y terminales de video, así como la capacidad de compartir y controlar los accesos a uno o varios discos magnéticos de gran capacidad, para efectos de consultas o movimientos. Para este tipo de tareas se sigue requiriendo una minicomputadora, o incluso un *mainframe*. Por otro lado, una (o varias) computadora personal sí ofrece una solución adecuada para las necesidades de planeación financiera de cada departamento de la fábrica, por ejemplo, ya que en ese caso se trata de tareas locales, desligadas de la operación global.

Existen sistemas que interconectan un conjunto de computadoras personales entre sí en forma de una red, y con ellos se ha hecho el intento de resolver problemas de procesamiento de datos e información que involucran a varios usuarios, pero en general el rendimiento obtenido es menor (y más caro) que el obtenido con sistemas diseñados ex profeso para esta labor.

Surge así una aparente paradoja: cada vez hay computadoras personales más y más poderosas (con microprocesadores de 32 bits y capaces de efectuar varios millones de instrucciones en un segundo), pero resulta que las aplicaciones para las que se podrían dedicar ya están ampliamente cubiertas por las de menor capacidad, por lo que ya no conviene continuar operando los nuevos modelos de computadoras personales con sistemas operativos que

atienden a un solo usuario. Esto significa que se llegó ya a una fusión entre las computadoras personales más recientes y parte de las que aún se llaman minicomputadoras. Así pues, poco a poco, las diferencias entre una computadora para un solo usuario y una para varios dejan de radicar en sus características físicas o electrónicas, y cada vez se desplazan más hacia las funciones desempeñadas por su sistema operativo, que es el conjunto de programas y sistemas con el que se opera. Esto puede observarse en la evolución de los sistemas operativos que han fungido como estándares en la línea de las computadoras tipo PC, que comenzaron siendo exclusivamente dedicados a un solo proceso y que ahora tienden a la atención concurrente de varias tareas.

Una computadora personal incluye componentes de cada una de estas categorías:

- Componentes electrónicos integrados (microprocesador, memoria, otros)
- Arquitectura global y canal
- Entrada/salida e interfaces gráficas
- Interfaces para dispositivos periféricos
- Sistema operativo

A continuación se mencionarán los términos principales que se manejan cuando se habla de cada uno.

Componentes electrónicos integrados

Como ya se mencionó, existen varias familias comerciales de microprocesadores, y a partir de 1986 han surgido los que actualmente son los más poderosos, de 32 bits. La tecnología de semiconductores más empleada es la que se conoce como CMOS (*Complementary Metal-Oxide Semiconductor*), y éstos y otros circuitos integrados se montan sobre placas de circuitos impresos, de una o varias capas. Últimamente, y debido a que cada vez se colocan más circuitos integrados en una placa, se ha desarrollado una técnica de montaje superficial, *Surface-Mounted Technology*, en la que los circuitos integrados no se insertan en la placa (también conocida como tarjeta) sino que se colocan (y sueldan) sobre la superficie, permitiendo una mayor densidad y funcionalidad.

También han surgido minúsculos circuitos integrados de memoria de gran capacidad, que contienen varios componentes montados superficialmente en forma de tiras delgadas, y que reciben el nombre de SIM (*Serial In-line Module*). En las nuevas computadoras, estos circuitos están reemplazando a los circuitos integrados de memoria usuales, de tecnología de empaque conocida como DIP (*Dual In-line Package*).

En algunas computadoras, en lugar de los circuitos integrados tradicionales se emplean ahora módulos con muy alta integración electrónica que se encargan de las funciones de varios circuitos, y que están formados internamente por grandes conjuntos de compuertas lógicas; reciben el nombre, en inglés, de *gate arrays*. A partir de ellos (y otras tecnologías afines) se ha hecho posible

que los diseñadores de microcomputadoras definan sus propios circuitos integrados, y entonces se habla de tecnología ASIC (*Application Specific Integrated Circuits*).

Arquitectura global y canal

Aunque dos computadoras empleen el mismo microprocesador pueden ser muy diferentes, tanto en características como en velocidad y costo, debido a que quizá no utilicen la misma filosofía de funcionamiento o de acoplamiento entre sus diferentes secciones electrónicas. A la forma en que está diseñada la estructura de la computadora completa (y no sólo de su procesador central) suele dársele el nombre de arquitectura, que ya hemos empleado. En las microcomputadoras, la arquitectura está organizada alrededor de una estructura común llamada *canal*. Las computadoras con el sistema operativo CP/M usaban un canal estándar conocido como *bus* S-100, que consistía en una tarjeta base a la que se conectaban las otras tarjetas (la del procesador, la de memoria, la de los controladores de disco, etc.). El *bus* S-100 fue abandonado con la introducción de la computadora PC, porque ésta utilizaba un canal llamado XT, en los modelos pequeños, y otro más elaborado, AT, en los modelos posteriores. Estos canales consisten en una tarjeta base principal que contiene al procesador, a la que se acoplan las otras tarjetas de la computadora mediante conectores especiales llamados *slots* (ranuras de expansión). Hay una versión mejorada del canal AT, llamada *Multibus*. Las máquinas que operan con procesadores de la familia Motorola 68000 utilizan el canal VME. Las nuevas computadoras Macintosh emplean un canal conocido como *NuBus*. Finalmente, la serie PS/2 de IBM incorpora uno nuevo, llamado *Microchannel*.

Entrada/salida e interfaces gráficas

Parte del éxito de las computadoras personales reside en su capacidad gráfica, muy mejorada con respecto a las primeras microcomputadoras. En una computadora tipo PC, la terminal de video se llama **monitor**, y está controlada directamente por el procesador, que mantiene en una memoria especial, dedicada exclusivamente para este propósito, una "imagen" electrónica de cada uno de los minúsculos puntos luminosos que se despliegan en la pantalla (hay varias decenas de miles), dedicando un bit para cada uno de los puntos en la pantalla de blanco y negro, y varios para el caso de monitores en color. De esta forma se obtienen efectos luminosos muy agradables y atractivos. Este método se llama *bit-mapped graphics*, y es rasgo distintivo de todas las computadoras personales, aunque su calidad y efectos varían porque existen varios estándares en cuanto a la resolución (número de puntos que se despliegan) y la gama de colores accesible. Las primeras computadoras personales empleaban una tarjeta llamada MDA (*Monochrome Display Adapter*) que ofrecía una resolución de 720 × 350 puntos, en blanco y negro, en pantalla. Ac-

tualmente es muy empleado el estándar monocromático *Hércules*, de alta resolución. Luego aparecieron las tarjetas para despliegue en color: CGA (*Color Graphics Adapter*), EGA (*Enhanced Graphics Adapter*), MGA (*Multicolor Graphics Array*) y VGA (*Video Graphics Array*), que ofrecen gamas de miles y miles de colores posibles, y resoluciones de varios cientos de miles de puntos. En general, para cada una de estas tarjetas hay que emplear un monitor especial, que no suele ser compatible con los que emplean las otras. Los monitores se pueden agrupar en dos grandes familias: analógicos y digitales. Por lo general, los analógicos son de mejor calidad y resolución que los digitales, porque disponen de una gama mucho mayor de colores para elegir.

Interfaces para dispositivos periféricos

Como la memoria central de las computadoras es limitada y volátil, se vuelve necesario utilizar dispositivos periféricos (es decir, no integrados a la unidad de procesamiento) de almacenamiento, con capacidades mayores a los 65 000 caracteres que cabían en la memoria de las computadoras con CP/M, o al millón que almacena una computadora PC avanzada, y para esto se emplean los discos magnéticos, que además ofrecen almacenamiento a largo plazo. En el capítulo 3 se describe su modo de funcionamiento, por lo que aquí tan sólo se mencionarán las diversas interfaces estándar que emplean las microcomputadoras para el manejo de los discos. Las primeras microcomputadoras utilizaban discos flexibles (*diskettes*) de baja capacidad, por completo incompatibles entre las diversas marcas y modelos, y luego se incluyeron pequeñas unidades de disco rígido, y entonces se hizo necesario poner orden en la diversidad de formatos diferentes de lectura y escritura que se empleaban. Para ello fue fundamental la gran presencia y peso en el mercado de las computadoras PC de IBM, ya que no fue sino hasta ese entonces que se obtuvieron estándares. Uno muy empleado, para control de discos rígidos, es el ST-506, aunque existen otros, como SCSI (*Small Computer System Interface*) y otro de más capacidad, llamado ESDI (*Enhanced Small Disk Interface*). Los controladores de disco rígido que utilizan estos estándares hacen posible conectar en una computadora unidades de marcas y capacidades diferentes, y las manejan de manera uniforme y a gran velocidad.

Sistema operativo

Como se dijo, los primeros sistemas operativos estándar en microcomputadoras fueron CP/M, por un lado, y el empleado por Apple, por otro. CP/M fue luego reemplazado por MS-DOS (en varias versiones), y en lugar de este último, en las nuevas computadoras PS/2 se emplea uno llamado OS/2, que ya incluye manejo de varias tareas concurrentes. En las computadoras grandes se habla de multiprogramación: manejo de varias terminales (es decir, usuarios) a la vez, mientras que en éstas se habla de ***multitasking***, manejo de varias tareas que un mismo usuario inicia desde la terminal. Las computa-

doras Apple siguen con su propia versión de sistema operativo, aunque los últimos modelos de la Macintosh trabajan con el sistema operativo Unix.

Otros sistemas operativos de uso relativamente común en las microcomputadoras son Concurrent DOS, CP/M-86, Business Basic (una combinación de sistema operativo con una versión del lenguaje BASIC orientado a los negocios), Unix (que se convirtió en estándar virtual para las microcomputadoras multiusuario), Xenix (una versión de Unix), Theos y Pick, aunque hay otros más.

La tendencia es que las microcomputadoras casi se conviertan en minicomputadoras y que sigan ofreciendo a sus múltiples usuarios (ya no sólo uno) un ambiente cómodo y con características atractivas, tanto visuales como de velocidad de proceso, y a un costo que (en términos de dólares) sigue descendiendo.

En un futuro se dispondrá de verdaderas supercomputadoras de tamaño reducido y con capacidades que, incluso hoy día, superan enormemente a las obtenidas por la ENIAC, creada apenas en 1947.

Palabras y conceptos clave

En esta sección se agrupan las palabras y conceptos de importancia que se estudiaron en el capítulo, con el objetivo de que el lector pueda hacer una autoevaluación que consiste en ser capaz de describir con cierta precisión lo que cada término significa, y no sentirse satisfecho hasta haberlo logrado. Se muestran en el orden en que se describieron o se mencionaron.

FENÓMENO ANALÓGICO (CONTINUO)
FENÓMENO DIGITAL (DISCRETO)
GENERACIONES DE COMPUTADORAS
MAINFRAME
MINICOMPUTADORA
MICROPROCESADOR
MICROCOMPUTADORA
DEPENDENCIA TECNOLÓGICA
CMOS
ASIC
CANAL

Ejercicios

1. Elabore un cuadro genealógico, lo más completo posible, de la historia de las computadoras.
2. Hay una "figura gris" en la historia de la computación; el Dr. Konrad Zuze en Alemania, antes y durante la Segunda Guerra Mundial, desarrolló máquinas que bien podrían considerarse como las primeras computadoras. Investigue sobre este tema.
3. Existe documentación muy interesante acerca de autómatas que resolvían diversos problemas y que, aunque no es posible denominarlos computadoras, representan un curioso antecedente histórico de los modernos sistemas de cómputo. Considérese, por ejemplo, el autómata para jugar ajedrez diseñado por el español Leonardo Torres en 1911, descrito en "Antique Mechanical Computers", de James M. Williams, publicado en tres partes por la revista *Byte*, julio-septiembre de 1978. Investigue sobre el tema. ¿Por qué estas máquinas no pueden llamarse computadoras?

4. Analice el impacto actual de las microcomputadoras personales desde una perspectiva amplia y objetiva.
5. ¿Sería conveniente enseñar computación en las escuelas primarias? Organice un debate sobre el tema.

Referencias para el capítulo 1

[DAVC78] Davis, N. C. y S. E. Goodman, "The Soviet Bloc's unified system of computers", *Computing Surveys*, Association for Computing Machinery, vol. 10, núm. 3, junio, 1978.
Interesante artículo publicado en una de las más prestigiosas revistas de computación en el mundo. Trata con cierto grado de detalle el desarrollo y estado de las ciencias de la computación en los países socialistas del área soviética.

[GOLH72] Goldstine, H. Herman, *The Computer from Pascal to von Neumann*, Princeton University Press, New Jersey, 1972.
En este libro se analiza el desarrollo e invención de las computadoras, desde sus orígenes remotos hasta la EDVAC. Incluye un interesante conjunto de fotografías de las grandes computadoras de los años iniciales. El señor Goldstine vivió algunas de las anécdotas descritas en el libro.

[LEVR82] Levine, Ronald, "Supercomputers", en *Scientific American*, enero, 1982.
Muy interesante artículo acerca de las computadoras especializadas para resolver enormes cantidades de cálculos, a velocidades de proceso de cientos de millones de instrucciones por segundo. Se presentan como ejemplos las máquinas STAR 100 (de CDC) y CRAY-1. Existe traducción al español.

[LUKH79] Lukoff, Herman, *From Dits to Bits. A Personal History of the Electronic Computer*, Robotics Press, Oregon, 1979.
Aquí se describe, desde una perspectiva de testimonio, el nacimiento de la computadora electrónica digital, pues el autor participó activamente en el desarrollo de la UNIVAC I. La introducción del libro fue elaborada por: "Dr. J. Presper Eckert y Dr. John W. Mauchly, inventores de la computadora electrónica".

[PYLZ75] Pylyshyn, Zenon (ed.), *Perspectivas de la revolución de los computadores*, Alianza Editorial, Col. Alianza Universidad, (núm. 119), Madrid, 1975.
En este compendio de artículos escritos por autores considerados pioneros en la historia computacional se pueden encontrar

descripciones de trabajos de científicos de la talla de Charles Babbage, Howard Aiken, Alan Turing, John von Neumann y Claude Shannon, así como reflexiones acerca del impacto de las computadoras y la computación por parte de investigadores de ciencias sociales como Margaret Mead y Herbert Simon.

[RALA76] Ralston, Anthony y C. L. Meek, (eds.), *Encyclopedia of Computer Science*, Petrocelli/Charter, Nueva York, 1976.
En más de mil quinientas páginas, doscientos autores tratan casi todos los temas imaginables sobre computación, en un nivel elemental o intermedio. Tiene una buena sección sobre historia de las computadoras.

[RUSB86] Russell, Bertrand, *Los problemas de la filosofía*, Nueva Colección Labor, Barcelona, 1986.
En este pequeño libro escrito en 1912, Bertrand Russell (quien además es uno de los grandes matemáticos del siglo XX) desarrolla una intensa e inteligente argumentación acerca de la necesidad de que exista la filosofía entre los humanos, y hace una apología final de extrema belleza.

[SLAR87] Slater, Robert, *Portraits in Silicon*, MIT Press, Massachusetts, 1987.
En este compendio se reseñan muchas de las personalidades más importantes de la computación, desde sus orígenes hasta la actual revolución de las microcomputadoras. Dedica varias páginas a cada uno de 31 investigadores que de una u otra forma han sido indispensables en el desarrollo de la ciencia y la técnica de las computadoras y su creciente impacto en la sociedad.

2

¿Cómo funciona una computadora?

2.1 Introducción

En este capítulo se estudiará con cierto detalle el *modus operandi* de una computadora digital electrónica, sin emplear ningún tipo de terminología electrónica o de ingeniería, porque el hilo conductor será, sencillamente, una idea descriptiva de los procesos que suceden dentro de la máquina, aun cuando no se conozca todavía su estructura interna, tema que se tratará en el capítulo 3.

El método consiste en indagar los pasos necesarios (y el orden que siguen) para efectuar operaciones sencillas sobre elementos de información, de modo que se llegue al resultado deseado, que será, por ejemplo, una sencilla operación aritmética sobre números enteros.

Para esto, surgirá la necesidad de definir con todo cuidado dos tipos de objetos: los datos y las operaciones o funciones que actúan sobre ellos. En realidad esta primera etapa no es difícil; cualquiera que haya usado una calculadora de bolsillo sabe cómo hacerlo. El problema interesante consiste en indagar la forma de *describir* los pasos efectuados, y de comunicárselo a la computadora a continuación. Y aquí aparece la necesidad de inventar el concepto de **programa**, y es también aquí donde se gesta la concepción de las modernas computadoras programables y de la teoría de la programación que permite el modelado de la realidad.

Las descripciones forman un programa

Como ejemplo se propone pensar en los pasos necesarios para realizar con una calculadora común la operación de sumar 5 + 7.

Está claro que para lograr esta suma hay que informar a la calculadora qué operaciones se desea hacer y sobre cuáles datos se van a aplicar. Normalmente se comunican los datos a la calculadora presionando las teclas que describen la operación por efectuarse. Este es el proceso con detalle:

1. Presionar la tecla "5" (con esto se avisa a la calculadora que debe guardar este número en alguna memoria temporal, hasta decidir lo que se hará con él).

2. Presionar la tecla " + " (ahora la calculadora traslada el "5" a un acumulador interno especial y está lista para recibir el segundo operando).
3. Presionar la tecla "7" (con lo que se hace la suma de manera interna en el acumulador; la calculadora mantiene el resultado internamente).
4. Presionar la tecla " = " (esto indica a la máquina que ha terminado la serie de operaciones, y que libere el resultado).
5. La máquina despliega el resultado: "12".

Si ahora se decidiera hacer la operación 5/7 habría que volver a iniciar el proceso desde el paso 1, cambiando solamente la tecla " + " del paso 2 por la tecla correspondiente a la división. En realidad, la calculadora "no se ha dado cuenta" de las operaciones que se han hecho, porque se han efectuado como resultado de un proceso externo a la máquina misma; es decir, todo ha dependido del orden de los pasos que se ha decidido efectuar con ella. Para que el proceso de la suma (o de la división, o cualquier otro) se realice de manera interna, habría que escribir un programa para que la máquina "entienda" lo que se desea hacer.

¿Qué es un programa? La definición más elemental es que un programa es un conjunto explícito de pasos a seguir para lograr un fin determinado. En este caso, lo que interesa es lograr la suma de dos números, para lo cual hay que definir a la máquina las siguientes **instrucciones**:

1. Observar el primer número.
2. Llevarlo al acumulador para sumarlo con el número que sigue.
3. Efectuar la suma usando este segundo número que ahora se observa.
4. Mostrar el resultado.

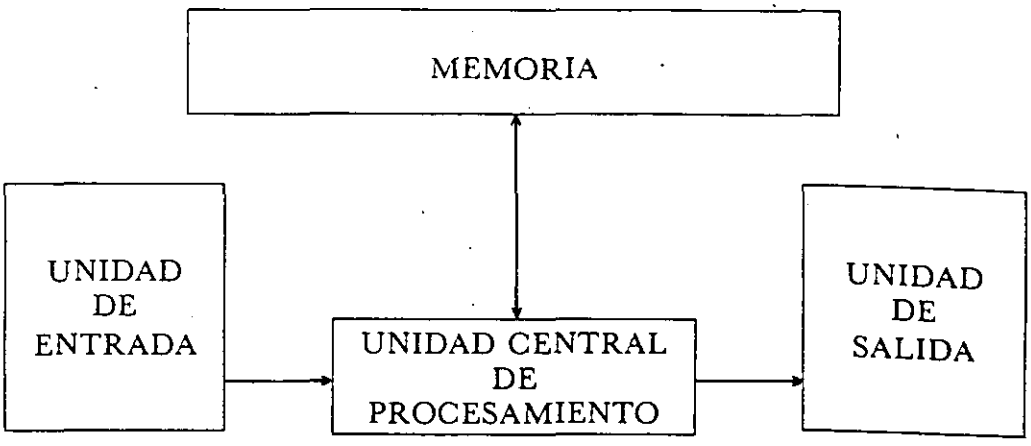
Pero hay todavía varios problemas por resolver para estar satisfechos con este programa. El primero es ¿dónde se almacenan los números que se desea que la máquina "observe"?, y luego, ¿dónde (y cómo) se almacenan las instrucciones del programa?

Para responder a estas dos preguntas se definirá lo que se conoce como el modelo de von Neumann. Además, si se resuelven dichas cuestiones, la calculadora se habrá convertido en una verdadera computadora.

2.2 El modelo de von Neumann

La idea central del modelo de computación propuesto por John von Neumann es almacenar las instrucciones del programa de una computadora en su propia memoria, logrando con ello que la máquina siga los pasos definidos por su **programa almacenado**

Una computadora de programa almacenado (que es otro nombre para una máquina que funciona con el modelo de von Neumann) tiene la siguiente configuración general, muy parecida, como se decía en el capítulo anterior, al diseño original de Charles Babbage, aunque no está basada en él:



Esquema básico de una computadora actual

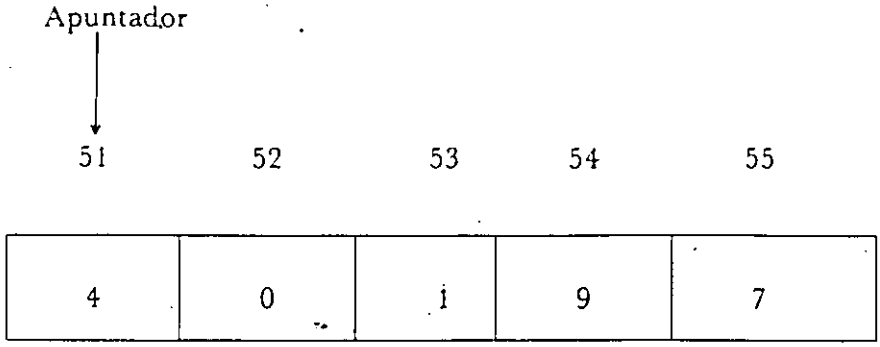
En este esquema se observan las relaciones estructurales que existen entre las diversas unidades que configuran la máquina, y que se emplean en prácticamente todos los modelos de computadoras. La unidad central de procesamiento (**UCP**, de aquí en adelante, aunque también es común referirse a ella como CPU, por sus siglas en inglés) contiene a la unidad aritmética y lógica (que hace los cálculos) y a la unidad de control.

Para poder operar bajo el modelo de von Neumann es necesario resolver el problema de comunicar a la computadora las operaciones por efectuar sobre los datos previamente almacenados en la memoria.

Se abordará el problema del almacenamiento de números recordando que la función de la memoria es guardar datos. Para nuestros propósitos, la memoria será un conjunto de **celdas** (o casillas) con las siguientes características: a) cada celda puede contener un **valor** numérico, y b) cada celda tiene la propiedad de ser direccionable, es decir, se puede distinguir una de otra por medio de un número unívoco que es su **dirección**. Esto implica que las celdas de la memoria tienen que estar organizadas de modo que faciliten la localización de cualquiera de ellas con un esfuerzo mínimo. La forma más sencilla de hacer esto es organizando las celdas en forma de **vector**, que no es más que un conjunto de celdas numerado secuencialmente.

Como se dijo, se puede hacer referencia a una celda por medio de su dirección. Se usará un **apuntador** para dirigirse a alguna celda cualquiera.

Un arreglo (que es otro nombre para un vector) en memoria se ve como sigue:



Cada celda tiene una dirección. Por ejemplo, la celda 51 contiene un 4.

Se dispone ya de una manera de almacenar (y recuperar) valores en la memoria por medio de una dirección unívoca. Es posible definir dos operaciones elementales sobre ella: *leer* el contenido de una celda y *escribir* un valor en una celda.

Si se supone que la memoria de una computadora es una especie de almacén atendido por un empleado que seguirá nuestras órdenes, éstos serán los pasos necesarios para poder efectuar las dos operaciones primitivas.

Operaciones sobre la memoria

Para leer:

A) Decidir cuál celda se va a leer (esto es, proporcionar su dirección).
B) Esperar un tiempo fijo para que el empleado vaya a la memoria y traiga el valor depositado en esa celda (la celda no pierde ese valor; sólo se trae una copia del dato y no el dato mismo).
C) Recoger ese dato y dar por terminada la operación de lectura.

Y para escribir:

A) Proporcionar al ayudante el dato que se desea depositar en una celda.
B) Proporcionar la dirección de la celda sobre la que se desea hacer la escritura del dato.
C) Esperar un tiempo fijo para que el empleado vaya a la memoria y deposite el dato en la celda designada, para dar por terminada la operación de escritura. (Si la celda en cuestión tenía ya un valor, éste se pierde, pues es reemplazado por el nuevo.)

Necesidad de la Codificación

Ahora hay que resolver el segundo problema: cómo almacenar las instrucciones en la memoria. Considerando lo anterior, habrá que encontrar una manera de hacer caber las instrucciones en las celdas. Esto lleva necesariamente al concepto de **codificación**. En efecto, si en las celdas de memoria

sólo caben números, entonces habrá que traducir las instrucciones a números para poder almacenarlas.

Para codificar las instrucciones se debe considerar cuántas y cuáles son las instrucciones disponibles y qué esquema de codificación se empleará.

El primer factor depende fundamentalmente de la capacidad de la unidad de control del procesador central para hacer operaciones; cuanto más compleja —y costosa— sea la unidad central de procesamiento, tanto mayor será el número de instrucciones diferentes que podrá efectuar. Después, se debe encontrar un código adecuado para que a cada instrucción definida corresponda uno, y sólo un, valor numérico. Para este segundo caso se usará una especie de diccionario electrónico (que forma parte de la unidad de control), que contendrá, por ejemplo, lo siguiente

Instrucción	Código interno
SUMA	57
RESTA	42
.	.
.	.
.	.

De aquí en adelante se empleará el nombre **lenguaje de máquina** para referirse al código que maneja la unidad central de procesamiento de la computadora.

Un primer programa

Ahora es posible escribir un primer programa completo, usando el modelo recién descrito. Se continuará con el problema de sumar 5 + 7.

Primera consideración: se requieren tres casillas, dos para los datos (5 y 7) y una para depositar el resultado. Se escogen las casillas 21, 22 y 23. (No hay ninguna razón especial para haberlas escogido; para nuestros fines, tres casillas cualesquiera son adecuadas.)

Segunda consideración: hay que definir con detalle las operaciones por efectuar y su orden, así como obtener una codificación adecuada (o sea, traducirlas a instrucciones para la máquina).

Tercera consideración: hay que introducir todos los datos (e instrucciones) en la memoria.

Analizando el programa de la página 30, se llega a la conclusión de que es necesario definir varias operaciones sobre la máquina. Se requiere, por lo pronto, una instrucción para llevar el contenido de una celda al acumulador (que es una celda especial, o **registro**, contenido en la UCP); otra para hacer la suma, y otra para devolver el contenido del acumulador a una celda de la memoria.

La forma de la instrucción para llevar el contenido de una celda al acumulador (que se llamará CARGA_Ac) es

```
CARGA_Ac dirección
```

donde CARGA_Ac es el nombre de la instrucción, y la dirección indica la celda de memoria cuyo valor se desea llevar al acumulador. Cabe aclarar que CARGA_Ac es el **nombre mnemónico** de la instrucción, pero que es necesario asignarle cierto código numérico interno. Sin importar ahora cuál sea éste, obsérvese que ocupará el contenido de una celda de la memoria. De la misma manera, la dirección será un número que ocupará un lugar en otra celda. Esto quiere decir que la instrucción CARGA_Ac ocupará *dos* celdas en la memoria: una para el *código* de la operación y la otra para la *dirección* a la que hace referencia.

En términos generales, habrá instrucciones que ocupen una, dos y hasta tres o más celdas de memoria.

Las demás instrucciones que se requerirán son

GUARDA_Ac dirección	que deposita el valor del acumulador en una celda de la memoria (ésta es la inversa de la anterior), y
SUMA dirección	que suma al acumulador el contenido de la celda de memoria descrita por la dirección.

Y el diccionario será, entonces,

Instrucción	Código interno	Longitud de la instrucción
CARGA_Ac	21	2
GUARDA_Ac	96	2
SUMA	57	2
RESTA	42	2

(Los códigos internos para el lenguaje de máquina que se escogieron son arbitrarios, aunque sí debe tenerse cuidado de usarlos consistentemente.)

Se escribirá el programa en forma tabular. En la parte izquierda del renglón se coloca la instrucción mnemónica seguida de la dirección a la que haga referencia (si se da el caso), luego se escribe su equivalente en el código interno extraído del diccionario y, por último, se describe brevemente el renglón (si se considera necesario).

Para lo que sigue, se supone que la celda 21 contiene un 5 y la celda 22 un 7, sin preocuparse por ahora de cómo se colocaron allí esos números.

He aquí el programa para sumar 5 + 7.

Instrucción	Dirección	Código	Comentarios
CARGA_Ac	21	2121	Se coloca el primer número en el acumulador.
SUMA	22	5722	Se efectúa la suma.
GUARDA_Ac	23	9623	El resultado queda en la casilla 23.
ALTO	--	70	

De este simplísimo programa se puede aprender varias cosas. Fue necesario inventar una nueva instrucción (ALTO) para lograr que la secuencia —cuando se ejecuta— llegue a un fin. Obsérvese que esta nueva instrucción ocupa una sola casilla de memoria, ya que no es necesario hacer referencia a alguna dirección.

Otra cuestión importante es la aparición de dos programas: uno escrito en lenguaje mnemónico (más fácil de reconocer para nosotros ya que es cercano al español), y otro —a la derecha— que está descrito en código numérico (el único que reconoce la computadora).

Se llamará **programa fuente** al primero y **programa objeto** al segundo. Esto es, el programa fuente es aquel que está escrito en un lenguaje similar al nuestro (pero inaccesible para la computadora), mientras que el programa objeto ya está traducido al código que la máquina reconoce. En este caso fue la misma persona la que escribió ambos programas; por lo general, será tarea del programador escribir el programa fuente, y la propia computadora lo traducirá a lenguaje objeto.

El programa objeto, entonces, es

```
2121 5722 9623 70
```

que, obviamente, es por completo ininteligible para un ser humano.

Otro aspecto fundamental es entender que este programa sirve para sumar cualquier par de números, siempre que residan en las casillas 21 y 22. Esto es realmente importante, pues significa, ni más ni menos, que es una especie de programa universal para sumar dos números, sin importar cuáles sean. Claro que esto no resulta impresionante por ahora, pero si se piensa en un programa universal para resolver cualquier ecuación algebraica de tercer grado, u otro para invertir cualquier matriz de orden 90 × 90, se apreciarán las ventajas de esta nueva herramienta.

Resta tan sólo introducir el programa objeto en la memoria de la computadora, para que pueda ejecutarse luego. Aquí es crucial elegir las casillas de la memoria que se utilizarán para almacenar el programa; esto es, en qué sección de la memoria se va a **cargar** el programa.

Se decidió hacerlo a partir de la celda 10 (se puede cargar a partir de cualquiera que esté desocupada, siempre que se esté seguro de que existen suficientes celdas secuenciales vacías).

Una vez cargado, el programa objeto se verá así:

21	21	57	22	96	23	70	...	05	07	??	....
10	11	12	13	14	15	16	...	21	22	23	....

Cada celda contiene un solo número de dos dígitos. La celda 23 contiene un número no especificado todavía —el resultado de la suma—, que se tendrá una vez ejecutado el programa.

Cada una de las dos primeras celdas contiene un número 21, pero en la primera, éste representa el *código* de la instrucción `CARGA_Ac`, mientras que el mismo 21 representa, en la segunda celda, la *dirección* 21. Obsérvese que una celda puede contener un mismo número que significará dos cosas diferentes, dependiendo del orden en que aparezcan con respecto al inicio.

Cuando se ha cargado el programa objeto a partir de la celda 10 de la memoria, hay que encontrar un procedimiento para lograr que la computadora comience la ejecución del mismo y poder así obtener los resultados deseados.

Aquí se hará un paréntesis para describir con cierto detalle cómo funciona la unidad de control del procesador central de la computadora.

La unidad de control

La función principal de la unidad de control de la UCP es dirigir la secuencia de pasos de modo que la computadora lleve a cabo un ciclo completo de ejecución de una instrucción, y hacer esto con todas las instrucciones de que conste el programa. Los pasos para ejecutar una instrucción cualquiera son los siguientes:

El ciclo de ejecución de la UCP

I. Ir a la memoria y extraer el código de la siguiente instrucción (que estará en la siguiente celda de memoria por leer). Este paso se llama **ciclo de *fetch*** en la literatura computacional (*to fetch* significa traer, ir por).
II. Decodificar la instrucción recién leída (determinar de qué instrucción se trata).
III. Ejecutar la instrucción.
IV. Prepararse para leer la siguiente casilla de memoria (que contendrá la siguiente instrucción), y volver al paso I para continuar.

La unidad de control ejecutará varias veces este ciclo de cuatro "instrucciones alambradas" a una enorme velocidad*.

Se llama así a estas instrucciones porque no residen en memoria, ni fueron escritas por ningún programador, sino que la máquina las ejecuta directamente por medios electrónicos, y lo hará mientras esté funcionando (mientras esté encendida). La ejecución de estos cuatro pasos (que forman un ciclo repetitivo) en una computadora es a razón de cientos de miles (o incluso millones) de veces por segundo.

Se ha definido ya el modelo de von Neumann. Ahora se pondrá a funcionar sobre nuestro pequeño programa de ejemplo (que ya está cargado en la memoria).

Pasos en la ejecución de un programa

Se describirán todos los pasos con detalle por única vez, para que el lector pueda estudiarlos con detenimiento hasta estar seguro de haberlos comprendido.

0. En virtud de que el programa comienza a partir de la celda número 10, se debe indicar a la unidad de control que esa celda contiene la primera instrucción. Esto se hace por medio de un apuntador (que forma parte de los circuitos electrónicos de la unidad de control) que recibe el nombre de **contador de programa (CP)**. Así pues, el primer paso debe consistir en apuntar a la casilla 10, y esto se representará por: CP ← 10.
(Obsérvese que este paso es externo, esto es, no forma parte del programa, sino que se tiene que hacer "desde afuera", para iniciar la operación de la computadora.)

1. La unidad de control ejecutará el paso I e irá a la casilla 10 para leer su contenido, que es 21.

2. La unidad de control ejecuta el paso II, con lo que decodifica el 21 recién leído y determina que se trata de una operación `CARGA_Ac`. En este momento sucede algo de primordial importancia: como la instrucción 21 tiene una longitud de dos celdas, una para el código (21) y otra (la siguiente) para la dirección de la celda cuyo valor se cargará en el acumulador (que en este caso —de casualidad— también es 21), la máquina deberá ajustar el valor del contador de programa para que éste apunte a esa celda siguiente (*i. e.*, la número 11).

3. La unidad de control ejecuta el paso III, con lo que efectivamente efectuará la operación de carga. Para esto, la computadora debe ir a la celda 11 y extraer su contenido, pero ahora ya no lo considerará como instrucción, sino como dirección, por lo cual irá a la celda 21 para extraer el valor que contenga (que es 5).

* Como se mencionó en el capítulo anterior, ésta es la diferencia principal entre hardware y software. El primer término denota todo lo referente a los circuitos electrónicos de una computadora, mientras que el segundo hace referencia a los programas, que no son parte física de la máquina, sino que residen en la memoria.

En este momento hay que tener cuidado para que no haya confusión: el primer 21 (el de la celda 10) es la instrucción CARGA_Ac; el segundo 21 (el de la celda 11) es la dirección de la celda de memoria cuyo valor se desea cargar en el acumulador. Esta instrucción completa, 21 21, puede leerse de la siguiente manera: cargar el acumulador con el valor que esté contenido en la celda cuya dirección aparece a la derecha de donde se está leyendo ahora. Conviene tener muy claro esto antes de seguir adelante.

4. La unidad de control ejecuta el paso IV, para luego ejecutar todo el ciclo de nuevo. Obsérvese que es un ciclo ilimitado, que sólo terminará cuando se ejecute la instrucción ALTO. En este caso, el contador de programa se hará igual a 12; esto es, apuntará a la celda número 12.

5. Se ejecuta (por segunda vez) el paso I de la unidad de control. Como CP = 12, se leerá esa celda, que contiene un 57.

6. Se decodifica esa instrucción, que es SUMA_Ac, por lo que el CP se prepara para apuntar a la siguiente celda. (Recuérdese que la instrucción SUMA_Ac ocupa dos celdas: una para el código de operación y otra para la dirección de la celda cuyo contenido se sumará al acumulador.)

7. Se ejecuta la instrucción 57, con lo que se añade el contenido de la celda 22 al acumulador (la dirección 22 reside en la celda 13, que es a la que actualmente apunta el contador de programa como resultado del paso anterior). Ahora el acumulador contendrá un 12 (o sea, 5 + 7).

8. El CP se actualiza para apuntar a la celda 14, en la cual (y no es casualidad) reside el código de la siguiente instrucción.

9. Se lee la celda 14 y se extrae su contenido: 96.

10. Se decodifica la instrucción, que es GUARDA_Ac, por lo que el CP se alista para apuntar a la siguiente celda, que contendrá la dirección de la celda en donde se guardará el contenido del acumulador.

11. Al ejecutarse esta instrucción se deposita el valor del acumulador (12) en la celda número 23, o sea, se deja el resultado de la suma en la celda que de antemano se había separado para tal fin.

12. La unidad de control regresa al paso I, no sin antes actualizar el contador de programa para que apunte a la celda 16, que es donde reside la siguiente instrucción.

13. Se lee la celda 16 y se extrae su contenido: 70.

14. Se decodifica esta instrucción, que es ALTO. El CP *no* se prepara para extraer un dato de la siguiente celda porque la instrucción 70 ocupa una sola celda.

15. Se ejecuta esta instrucción, lo que detiene a la unidad de control y a la máquina. De esta manera se rompe el ciclo de los cuatro pasos.

Si el lector siguió estos dieciséis pasos con cuidado (con ayuda del diagrama de la página 36) habrá aprendido varias cosas, entre las que sobresalen las siguientes:

- Dado el contenido de una celda, la computadora no puede distinguir si se trata de una instrucción o de un dato o dirección.
- Debido a lo anterior, es responsabilidad de quien maneja la máquina indicarle cuál es la celda en donde comienza el programa (esto se hizo

por medio del paso 0, que se describió como externo al programa). Más adelante se verá que puede ser la propia computadora, por medio del sistema operativo, la que se encargue de esta tarea.

- Una vez que el contador de programa apunta a la celda que contiene la primera instrucción, el resto del proceso ocurre de manera automática e invisible para el programador. Esto se debe a los ajustes internos que se hacen al CP (en el paso II) que, a su vez, dependen de la longitud de la instrucción que se está ejecutando.

Sistema binario

En una computadora real las celdas no contienen números de la forma que se ha descrito aquí (lo cual se hizo por comodidad), sino que los almacenan en forma binaria. El **sistema binario** es casi el más sencillo que existe para representar números. Este "casi" es porque la manera más simple para representar, por ejemplo, el número seis, es con el sistema unario:

111111 o bien XXXXXX

Pero habría graves problemas al intentar representar el número diez millones, por ejemplo.

En el sistema binario se escoge el número dos como la base de la numeración (así como en el unario se escogió el uno), y se representa un número cualquiera descomponiéndolo en las sucesivas potencias de dos que, sumadas, den como resultado el número en cuestión. Por ejemplo, para escribir el número 60 en binario habría que descomponerlo de la siguiente manera: una vez dos elevado a la quinta potencia, más una vez dos elevado a la cuarta potencia, más una vez dos elevado a la tercera potencia, más una vez dos elevado al cuadrado, más cero veces dos elevado a la primera potencia, más cero veces dos elevado a la cero potencia. En términos más sencillos sería:

$$60 = 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0,$$

porque

$$60 = 32 + 16 + 8 + 4.$$

Más sencillo aun:

$$60 = (111100)_2.$$

En el sistema decimal común y corriente se omite decir que la base es diez, cosa que sí se dijo para el caso 111100 encerrándolo entre paréntesis y poniendo el 2 como subíndice.

Estrictamente hablando,

$$(60)_{10} = (111100)_2.$$

Un resultado teórico debido al matemático e ingeniero norteamericano Claude Shannon especifica que el sistema binario es suficiente para representar cualquier cantidad de información. Y esta propuesta constituye la base del campo nuevo llamado **teoría de la información**, del que se dará un ejemplo sencillo. Si se considera el problema de averiguar la edad de una persona, en lugar de preguntárselo directamente, podría intentarse encontrar un método general y, al mismo tiempo, definir ciertos conceptos nuevos. Se podría preguntar: ¿tienes menos de treinta años?, y la respuesta sólo podría ser sí o no. A partir de esa respuesta podrían hacerse más "preguntas binarias" (preguntas cuya contestación sea sí o no), hasta encontrar el dato buscado. Se dice que la **cantidad de información** que un dato contiene se mide por el número mínimo de preguntas binarias requeridas para averiguarlo con exactitud.

Este número mínimo fue acotado y medido por Shannon, pero ya no tiene por qué preocuparnos aquí. Lo que sí interesa es darle el nombre de dígito binario, o **bit**, a ese posible cero o uno que será, de aquí en adelante, la **unidad mínima de información**. El artículo [BHAR87] contiene una explicación introductoria a la teoría de la información y a los resultados de Shannon.

Unidades de medida

Los bits suelen trabajarse en grupos de ocho. Cada uno de estos pequeños grupos recibe el nombre de **byte**. Los bytes, a su vez, forman grupos de 1024, y cada uno de estos grupos se conoce como un **kilobyte (KB)**. Se escogen estos números porque $8 = 2^3$ y $1024 = 2^{10}$.

Finalmente, cerca de un millón de bytes constituyen un **megabyte (MB)**. El número exacto es $2^{20} = 1\ 048\ 576$ bytes.

Es importante tomar en cuenta que prácticamente todas las computadoras comerciales (sin importar la marca o el tipo) siguen el modelo recién descrito. Esto no quiere decir, sin embargo, que la programación se haga de manera tan burda como la mostrada; de hecho, ningún programador trabaja así, sino que utiliza las facilidades que la programación de sistemas le proporciona, por medio de ensambladores, compiladores y sistemas operativos, que se estudiarán en los capítulos siguientes.

Palabras y conceptos clave

En esta sección se agrupan las palabras y conceptos de importancia que se estudiaron en el capítulo, con el objetivo de que el lector pueda hacer una autoevaluación que consiste en ser capaz de describir con cierta precisión lo que cada término significa, y no sentirse satisfecho hasta haberlo logrado. Se muestran en el orden en que se describieron o se mencionaron.

PROGRAMA	VALOR	LENGUAJE DE MÁQUINA
INSTRUCCIONES	DIRECCIÓN EN MEMORIA	PROGRAMA FUENTE
MODELO DE VON NEUMANN	VECTOR	PROGRAMA OBJETO
PROGRAMA ALMACENADO	APUNTADOR	CARGA EN MEMORIA
CELDA	CODIFICACIÓN	CICLO DE *FETCH*

HARDWARE
SOFTWARE
CONTADOR DE PROGRAMA
SISTEMA BINARIO
CANTIDAD DE INFORMACIÓN
BIT
BYTE

Ejercicios

1. ¿Qué sucede si un programa escrito en lenguaje de máquina no termina con la instrucción ALTO?
2. ¿Qué pasa si, equivocadamente, se hace CP ← 11 en lugar de CP ← 10 en el programa de la página 36?
3. ¿Por qué un KB son 1024 bytes, y no mil, que sería más cómodo?
4. Escriba un programa en lenguaje de máquina (usando el diccionario interno definido en el texto) para a) sumar tres números cualesquiera, y b) restar un número al resultado anterior.
5. Invente y codifique las instrucciones necesarias para que, una vez añadidas al diccionario interno de la máquina, se pueda resolver lo siguiente:
   A) Obtener el promedio de cuatro números almacenados en la memoria. Considere únicamente operaciones sobre números enteros. Por lo tanto, el promedio que obtendrá será también un número entero.
   B) Encontrar el mayor entre dos números almacenados en la memoria y dejarlo en el acumulador.
   C) Determinar si tres números que ya están almacenados en la memoria pueden representar las longitudes de los lados de un triángulo rectángulo y, si es así, calcular su perímetro. Por ejemplo, los números 3, 4 y 5 sí representan los lados de un triángulo rectángulo porque $3^2 + 4^2 = 5^2$. Los números 7, 24 y 25 representan otro ejemplo*. (Debe tenerse cuidado con el tamaño de los números, porque en una celda no pueden caber números de longitud arbitraria, sino que hay un límite, que depende de cada máquina. Es posible que al elevar un número al cuadrado ya no quepa en una celda.)
6. En la memoria están almacenados tres precios de un mismo producto. Escriba un programa en lenguaje de máquina que encuentre el precio menor y que calcule las diferencias numéricas de los otros dos con respecto a éste. Utilice valores enteros.
7. Existen varios métodos para convertir un número de representación decimal a binaria y viceversa. Escoja alguno y describa sus pasos con detalle, intentando ponerlo en forma de un programa. No es necesario escribirlo en lenguaje de máquina, pero sí debe estar especificado con el mayor detalle posible.

* Hay un número infinito de enteros que cumplen con la propiedad de que la suma de los cuadrados de dos de ellos es igual al cuadrado de un tercero, pero sólo hay 16 juegos de enteros menores que 100 que no tienen factores comunes. Está claro que se está hablando del teorema de Pitágoras, pero no es tan sabido que los sumerios ya lo conocían, siglos antes del filósofo, y en una tableta con caracteres cuneiformes aparece que la suma de los cuadrados de 4961 y 6480 es igual al cuadrado de 8161. Para información sobre el tema de estas ecuaciones, llamadas diofantinas, que ocupan un lugar clásico dentro de la teoría de los números, véase el artículo "Fermat's Last Theorem", de Harold M. Edwards, publicado en la revista *Scientific American*, octubre de 1978, o el curioso libro *Algebra recreativa*, de Yakov Perelmán, Editorial Mir, Moscú, 1978.

Referencias para el capítulo 2

[BHAR87] Barath Ramachandran, "Information Theory", en la revista *Byte*, diciembre, 1987.
Artículo que explica los fundamentos de la teoría de la información y maneja los resultados centrales de la teoría de Shannon. El artículo inicia con la observación de que el libro original de Shannon y Weaver sobre la teoría matemática de la comunicación es un "clásico de la ciencia del siglo XX".

Las referencias [GOLH72], [LUKH79] y [RALA76] del capítulo anterior se aplican también para éste. Un estudio más concienzudo puede encontrarse en las referencias del capítulo de programación de sistemas.

El libro *Organización de computadoras*, de Hamacher, Vranesic y Zaky, McGraw-Hill, México, 1987, contiene algunos capítulos que pueden ser de interés en este nivel, aunque en general se trata de un texto que requiere conocimientos en ingeniería electrónica.

3

Descripción funcional de un sistema de cómputo

Se llama *sistema de cómputo* a la configuración completa de una computadora, junto con unidades periféricas y con la programación de sistemas que la hacen comportarse como un todo coherente.

Está claro que el modelo de von Neumann implica que la memoria sea independiente del procesador y de la unidad de control. Se verá ahora cómo "armar" un sistema de computación moderno. Para esto es necesario analizar las unidades por separado, para luego hacer la integración.

Se seguirá un método descriptivo, en el que importa más la razón de ser de los elementos estudiados que su morfología interna. Interesan estas máquinas no desde un punto de vista operativo o ingenieril, sino en tanto sirvan de apoyo para nuestras finalidades conceptuales.

3.1 El procesador central

El conjunto que forman la unidad de control y la unidad aritmética y lógica se llama procesador central o unidad central de procesamiento, UCP. Sus funciones consisten en leer y escribir contenidos de las celdas de memoria, llevar y traer datos entre celdas de memoria y registros especiales (por ejemplo el acumulador), y decodificar y ejecutar las instrucciones de un programa.

El procesador es, pues, el "corazón" de la computadora. De él dependen las demás funciones del sistema integrado, y es el que controla todas las operaciones que la máquina realiza.

Como en todo sistema complejo donde interactúan muchos componentes, una computadora requiere una organización jerárquica para funcionar. En este caso la organización consiste en distribuir las tareas entre subsistemas diversos que reportan sus actividades al procesador central por medio de **interrupciones**. (Consúltese el artículo [ATKT79] para una explicación general sobre este tema.)

Normalmente, la UCP ejecuta a enorme velocidad los cuatro pasos descritos en el capítulo anterior (lectura en memoria, decodificación, ejecución, ajuste del CP). Cuando es necesario hacer una operación especial sobre alguno de los subsistemas externos (una lectura en disco, por ejemplo), la UCP da la orden y continúa la ejecución del programa. Cuando el subsistema termina lo que le fue encargado, manda una interrupción a la UCP para que ésta le indique qué otra operación especial (si la hubiera) hay por ejecutar.

La operación de la UCP está controlada por un **reloj maestro de tiempo real**, que es el que le indica cada cuándo se debe iniciar una nueva operación. En términos generales, será este reloj el que determine la velocidad de operación del procesador. Como ilustración se puede decir que una microcomputadora común está controlada por un reloj con una frecuencia de 8 a 20 MHz (millones de ciclos por segundo), mientras que las máquinas más grandes tienen osciladores con frecuencias de 20 ó 40 MHz. El circuito que actúa como reloj envía impulsos de control (para comenzar la ejecución de las operaciones y sincronizarlas) a razón de 10 millones por segundo para el caso del microprocesador 68010, como se describe en [CRAW86].

Sin embargo, no hay que confundir la frecuencia a la que opera el reloj con la cantidad de instrucciones que el procesador puede ejecutar, ya que son necesarios varios ciclos del reloj para hacer los cuatro pasos que requiere cada instrucción, y el número exacto depende de la complejidad de cada instrucción del lenguaje de máquina. Existen varias unidades de medida de la velocidad de un procesador, que tienen distintos niveles de significancia. La primera, la ya mencionada frecuencia del reloj, especifica tan sólo la cantidad de veces que la unidad de control recibe impulsos eléctricos en un segundo, y no es directamente relacionable con la velocidad de proceso de los programas en general, sino sólo de los componentes que constituyen el lenguaje de máquina. Una medida más cercana a los programas del usuario se conoce como **MIPS** (*Million Instructions per Second*) y se refiere a la cantidad promedio de instrucciones del lenguaje de máquina que la computadora ejecuta en un segundo. Una minicomputadora procesa normalmente a razón de entre 0.5 y 2 MIPS, y una gran máquina puede alcanzar 15 ó 20. Finalmente, existe otra medida, más estricta, llamada **FLOPS** (*Floating Point Operations per Second*), que se refiere a la cantidad de instrucciones aritméticas de punto flotante (es decir, operaciones aritméticas con números con punto decimal) que se pueden ejecutar en un segundo. Una de las llamadas supercomputadoras es capaz de procesar a la asombrosa velocidad de 600 megaflops (millones de FLOPS).

Unidades de medida de velocidad del procesador

Integrada al procesador existe una serie de celdas (análogas a las de la memoria) que se utilizan con mucha frecuencia y que, por ende, no están en la memoria sino que forman parte de la UCP. Estas celdas reciben el nombre de **registros**. Un procesador puede tener una decena o dos de ellos, rara vez más. Un registro muy importante, que ya se ha empleado en el texto, es el acumulador.

La unidad aritmética y lógica de la UCP, como su nombre indica, se encarga de efectuar las operaciones relacionadas con los cálculos numéricos y simbólicos. Una unidad típica sólo es capaz de realizar un número reducido de operaciones muy elementales, aunque a gran velocidad. Las operaciones que estas subunidades pueden efectuar son: suma y resta de dos números de punto fijo, multiplicación y división de punto fijo (no todos los procesadores tienen esta capacidad), manipulación de los bits de los registros y del acumulador (operaciones lógicas **AND**, **OR**, **NOT**), y comparación del contenido de dos registros (para averiguar si los números que contienen son iguales, o cuál es mayor).

Prácticamente ningún procesador tiene la capacidad de efectuar operaciones más complejas que éstas, lo que significa que, por ejemplo, para elevar un número a una potencia hay que usar un programa especial. Todas las computadoras proporcionan a los usuarios bibliotecas de programas y funciones matemáticas para efectuar estos cálculos, y lo hacen "armando" las funciones complejas con base en las operaciones elementales que la unidad aritmética y lógica sí es capaz de efectuar.

Es más, en opinión de algunos investigadores, las unidades centrales de procesamiento ya son demasiado complejas, y entonces ha surgido una corriente en sentido inverso, que pide que los procesadores sean sencillos pero muy rápidos, y que ha desembocado en un tipo especial de arquitectura (es decir, de diseño) en la que el lenguaje de máquina consta de unas cuantas —y muy sencillas— operaciones, y que se conoce como **RISC** (*Reduced Instruction Set Computer*). Ya hay varias computadoras comerciales que emplean este tipo de tecnología, con buenos resultados.

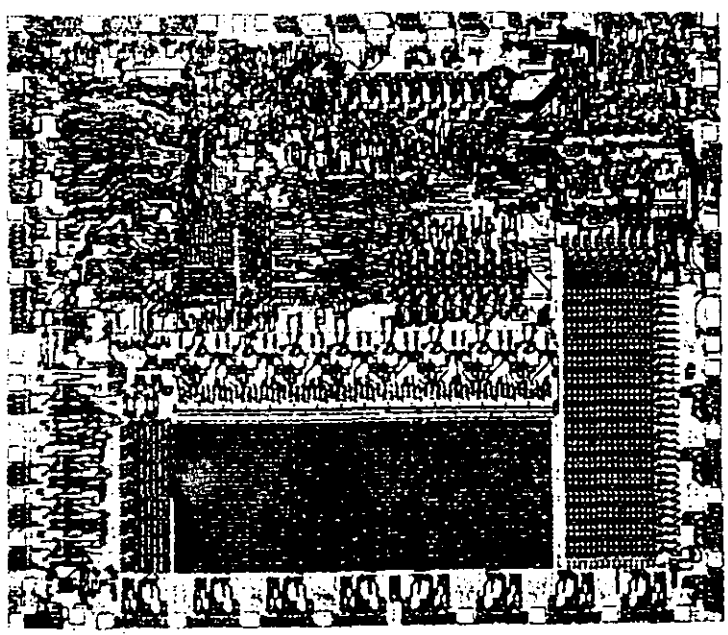
En principio, la potencia de cómputo de un procesador está dada en términos de la cantidad de bits que puede manipular en una sola operación. Es decir, el hecho de que un procesador pueda trabajar con operaciones y números de 16 bits le da una gran ventaja en velocidad y flexibilidad de operación con respecto a uno diseñado para trabajar sólo con 8 bits en paralelo. De esta forma, las computadoras pueden clasificarse por lo que se conoce como **tamaño de palabra**, esto es, la cantidad de bits que el procesador puede manejar a la vez.

Las máquinas con procesador de tamaño de palabra de 8 bits forman una familia de microcomputadoras, que están siendo reemplazadas por las de 16 bits, como se mencionó en el capítulo 1. Las minicomputadoras suelen tener procesadores de 16 bits, y algunas ya incluyen unidades de 32 bits. Las grandes computadoras procesan grupos de 32 o hasta 64 bits, y hasta la fecha prácticamente todas emplean procesadores que requieren decenas o más de circuitos integrados (es decir, aún no existen microprocesadores de más de 32 bits).

Es posible que un procesador logre la ejecución de instrucciones que no forman parte de su "diccionario electrónico" (o incluso que simule a otro procesador diferente) mediante una técnica (mitad electrónica, mitad de programación) llamada **microprogramación**. Este es un tema avanzado que no se tratará aquí. Los artículos [CLIB79] y [PATD83] contienen una explicación elemental de este importante concepto, usado para que la UCP ejecute operaciones complejas por medio de agrupaciones de los llamados *micropasos*. El libro [SIED82] es la referencia más amplia sobre arquitectura de computadoras, y ahí también se tratan con amplitud estos temas.

Existen además procesadores de diseño especial que sirven exclusivamente para hacer operaciones aritméticas más complejas que las ya descritas. Estos procesadores numéricos de punto flotante sí son capaces de elevar números a potencias, calcular logaritmos, exponenciales y otras funciones trascendentes, a la vez que efectúan las operaciones aritméticas elementales con gran precisión. Para que una computadora incluya uno de estos procesadores se requiere que el procesador central lo adopte y lo ponga a funcionar bajo la modalidad conocida como procesamiento "*amo-esclavo*". En este tipo de configuración, el procesador central lleva (como siempre) el control de todas las operaciones por realizar y pasa el control al procesador numérico cuando detecta la aparición de una de esas operaciones complejas; el procesador numérico la ejecuta y devuelve el control al amo. Luego de esto se desactiva y se mantiene así mientras no reciba la orden de realizar un nuevo trabajo.

El conjunto de operaciones de máquina que puede ejecutar un procesador es, como se ha dicho, limitado. Generalmente está dividido en grupos de operaciones afines. Para el caso del microprocesador INTEL 80286 (procesador de 16 bits), éstas son un poco más de 130, divididas en siete grupos (de transferencia de datos, aritmética, lógica, de manipulación de cadenas, de flujo



Un microprocesador, grandemente ampliado

de control, instrucciones de alto nivel, y de entrada/salida y control) (véase el manual [INTC87]).

Dicho microprocesador consiste en un circuito integrado de alta densidad (como los descritos en el artículo [VACA75]) donde, en un espacio menor que un centímetro cuadrado, existen más de 150 000 transistores y otros elementos electrónicos microscópicos "grabados" en una tableta de silicio. Requiere tan sólo de una fuente de potencia de cinco volts, lo que significa que puede operar perfectamente con una simple batería. La tecnología empleada para producir este tipo de microcircuitos recibe el nombre genérico de integración a muy alta escala (VLSI, por sus siglas en inglés).

El 80286 tiene mayor poder de cómputo que la ENIAC, que menos de treinta años antes requería 18 000 bulbos y ocupaba varias decenas de metros cuadrados. Pocas veces la humanidad ha contemplado avances tecnológicos tan extraordinarios y en tan poco tiempo.

Cuando se hable de sistemas operativos, en la sección 4.6, se volverá a mencionar el procesador central y se explicará cómo se puede controlar un aparato tan rápido y complejo como éste.

3.2 La memoria central

En este conjunto —generalmente grande— de celdas direccionables es donde la computadora almacena toda la información (datos y programas) que utilizará mientras esté encendida. Cualquier instrucción que el procesador efectúe deberá necesariamente residir en la memoria central, ya que es ahí donde la UCP buscará la siguiente instrucción, como parte del paso I del ciclo descrito en el capítulo anterior.

Las computadoras de la primera generación se caracterizaban por disponer de muy pocas celdas de memoria, pues éstas eran costosas y difíciles de construir. No obstante, a medida que avanzaba la electrónica digital, fue cada vez más factible agrupar grandes cantidades de celdas. La tecnología de las memorias de la primera y segunda generaciones estuvo dominada por las memorias de ferrita (*core memory*, en inglés). Esta denominación hace referencia a la manera en que estaban construidas: cada celda de memoria consistía en un grupo de ocho casi microscópicas rondanas de ferrita, atravesadas por cuatro alambres. El conjunto de varios miles de estas rondanas formaba un verdadero tejido que se ensamblaba a mano. Cada rondana actuaba como un electroimán, porque al pasar una corriente eléctrica enmedio de alguna de ellas se magnetizaba y podía, de esta manera, representar la presencia o ausencia de un bit de información. Esto es, si la ferrita estaba magnetizada, representaba un 1 lógico; de otro modo, representaba un 0 en el sistema binario. Es fácil comprender que así es posible configurar un sistema digital, pues en principio basta con asignar cada uno de los estados discretos del sistema a alguna combinación de ceros y unos.

En las máquinas de la tercera generación y las posteriores, las ferritas han sido reemplazadas por memorias de semiconductores, fabricadas con circuitos integrados, a base de microtransistores. La ventaja de estas memorias sobre las anteriores es que se pueden construir por métodos industriales (y no

manuales), con las consiguientes ventajas en precio y cantidad. Todavía en 1970 una computadora se consideraba grande si disponía de 20 000 celdas de memoria (de ferrita); quince años después es común que hasta una microcomputadora tenga alrededor de 250 000, y no sorprende encontrar máquinas con tres millones o más de celdas de memoria de semiconductores.

Las memorias de semiconductores operan en dos configuraciones: estáticas y dinámicas. Las primeras almacenan la información mientras estén alimentadas con corriente eléctrica, en tanto que las segundas requieren circuitos de "refrescamiento", que reescriben la información que contiene cada celda a razón de cientos de veces por segundo. Esto puede parecer raro, pero los modernos circuitos integrados de alta velocidad se encargan de que esta función se realice de tal forma que ni el procesador ni, por supuesto, el usuario se enteren.

El parámetro más importante en una memoria es su velocidad de acceso, que mide el tiempo transcurrido desde que el procesador central pide la información contenida en una celda cualquiera hasta que ésta puede ser leída (o escrita). Los tiempos de acceso de las memorias de semiconductores se miden en unidades de millonésimas de segundo.

Además, las memorias electrónicas verifican constantemente que la información almacenada no se altere o degrade, por medio de una técnica conocida como **detección de paridad**, que se explica más adelante en la sección dedicada a las memorias secundarias.

El nombre genérico de estas memorias (estáticas o dinámicas) es **RAM** (*Random Access Memory*, memoria de acceso aleatorio).

Una desventaja de éstas con respecto a las memorias de ferrita es que los circuitos integrados pierden la información que tenían almacenada cuando se interrumpe la alimentación eléctrica. Esto obliga al diseño de memorias no volátiles, en las que se graba información que ya no se pierde. Estos nuevos tipos de circuitos reciben el nombre genérico de **ROM** (*Read Only Memory*, memoria sólo de lectura), y se fabrican en varias configuraciones (PROM, EPROM, EEPROM), de acuerdo con la mayor o menor facilidad para regrabarlas (aunque en general únicamente se emplean para lectura, y son grabadas por el fabricante de la computadora, no por el usuario).

En términos generales, los programas que residen en una memoria tipo ROM se conocen, en inglés, como *firmware*, que representa un intermedio entre los programas normales (software) y los circuitos electrónicos (hardware). La separación entre hardware y software se ha borrado poco a poco debido sobre todo a la aparición de tecnologías de "hardware programable", que reciben el nombre genérico de **PAL** (*Programmable Array Logic*) y que consisten en circuitos configurables por el diseñador (o, a veces, por el usuario), y que permiten una flexibilidad que antes no existía.

Nuevas tecnologías de diseño electrónico

Las nuevas computadoras consisten a veces en unos pocos circuitos integrados de este tipo, que reemplazan a varias decenas de los normales. En inglés reciben nombres como *gate arrays* o ASIC, y se usan para el diseño de procesadores y arquitecturas completas.

Existe otro tipo de memoria que funciona con técnicas magnéticas especiales; durante algunos años se creyó que este tipo de memoria reemplazaría

por completo a las memorias de semiconductores, pero por motivos de economía de mercado esto aún no se ha logrado. En el artículo [HALI79] se describen estos circuitos, llamados **memorias de burbujas**.

3.3 Unidades de entrada y salida

Un procesador se comunica con el exterior por medio de **interfaces** que permiten la entrada y salida de datos del procesador y la memoria; ésta es la única manera de que el procesador se comunique con el entorno exterior a la computadora, pues es necesario emplear dispositivos de interfaz que hagan llegar la información de los usuarios hacia el procesador central, así como que les muestren los datos ya procesados.

Unidades de entrada

Las unidades de entrada más comunes son las lectoras de tarjetas (que casi han desaparecido) y las **terminales de video** (o pantallas). Las unidades de salida más usuales son las impresoras y las terminales de video. Existe gran diversidad de modelos de terminales de entrada/salida, pero la mayoría utiliza los dos mismos elementos que permiten la comunicación entre el humano y la máquina: un teclado (como el de una máquina de escribir) para comunicarse con la máquina, y una pantalla de video (como la de un televisor) donde la computadora escribe sus mensajes. Por lo común, en la pantalla aparece un carácter luminoso especial —llamado **cursor**— que sirve para indicar dónde aparecerá el próximo mensaje.

A veces se reemplaza la pantalla de video con una hoja de papel, y en tal caso se habla de un teletipo, pero esta práctica ha caído en desuso.

En lo que respecta a las unidades exclusivamente de entrada, hay que mencionar en primer lugar las **lectoras de tarjetas**. El origen de las tarjetas perforadas se remonta al siglo XVIII, cuando fueron inventadas para automatizar los telares mecánicos, reciente aportación de la Revolución Industrial. Para comunicarse con la computadora por medio de tarjetas, se codifican los caracteres que van a transmitirse, como orificios en el papel. Esto se logra por medio de la **perforadora** de tarjetas, que tiene un teclado similar al de una de máquina de escribir y produce una tarjeta por cada renglón de texto deseado. Como se dijo, estas unidades casi han desaparecido, fundamentalmente, porque las computadoras actuales se comunican de manera interactiva con el usuario (es decir, mediante un diálogo), para lo cual se requiere una unidad que permita la comunicación bidireccional (entrada/salida), no una unidad de entrada exclusivamente. La opción más adecuada para este fin es la terminal de video ya mencionada.

Unidades de salida

Las unidades exclusivamente de salida están representadas por una amplia gama de **impresoras**, que van desde las sencillas y relativamente lentas hasta impresoras computarizadas de muy alta velocidad. Las impresoras lentas por lo general funcionan con un mecanismo parecido al de una máquina de escribir eléctrica común, y son capaces de imprimir hasta diez caracteres por segundo, lo que significa que llenan una hoja tamaño carta en aproximadamente dos minutos. Las que siguen en velocidad imprimen a razón de cuarenta hasta trescientos caracteres por segundo y, en general, utilizan un me-

canismo de generación de cada carácter por medio de un conjunto de puntitos de tinta, que recibe el nombre de **matriz**. La calidad de la letra impresa no es muy buena, pues los puntitos que forman cada carácter son visibles y hacen la letra menos legible que la de una máquina de escribir común. Por su precio relativamente bajo, usualmente están asociadas con las micro y minicomputadoras. Una de estas máquinas puede llenar una hoja tamaño carta en treinta segundos de trabajo.

Las impresoras para computadoras grandes son capaces de producir textos de calidad comparable a la de una buena máquina de escribir, a razón de trescientas hasta mil líneas por minuto, lo que permite llenar una página tamaño carta en pocas decenas de segundos. En estas máquinas, los tipos están montados en una cadena que gira a gran velocidad, por lo que reciben el nombre de **impresoras de cadena.**

Existen enormes impresoras (a veces más costosas que la computadora misma), capaces de imprimir varias decenas de miles de líneas por minuto, esto es, imprimen una hoja tamaño carta en un segundo o menos. La complejidad de estos equipos es tal que generalmente están controlados por una computadora dedicada exclusivamente a ellos. El mecanismo de impresión es por medio de microscópicas gotas de tinta que un "cañón" lanza hacia el papel, para que dibuje cada uno de los caracteres de impresión. En algunos casos este flujo está controlado por un rayo láser que lo guía hacia su destino final en la hoja. Su velocidad es tal que, por ejemplo, si se requiere obtener un documento con varias copias, resulta más barato y rápido imprimir originales que obtener copias por otro medio (para una descripción véase el artículo [KUHL79]).

Obtener copias por otro medio

Cada vez son más comunes unas impresoras de mediana velocidad y capacidad que imprimen texto y gráficas de excelente calidad, y cuyo principio de funcionamiento es similar al de las fotocopiadoras, con la diferencia de que en estas impresoras es un rayo láser el que graba temporalmente la imagen por reproducir en el mecanismo entintador. Se espera que su precio continúe bajando, y tal vez pronto se conviertan en las impresoras usuales en aplicaciones que no requieren gran volumen de impresión. Estas **impresoras de láser** (y el software apropiado) han dado lugar al nacimiento de los llamados sistemas de edición por computadora, en los que una microcomputadora dotada de una terminal de graficación y de una impresora de láser es capaz de producir material gráfico y textos comparables en calidad y versatilidad a los que se obtienen en una pequeña imprenta. Los programas que sirven para estos fines se conocen como **procesadores de palabras**. Aunque cada procesador de palabras tiene características particulares todos son capaces, al menos, de alinear el texto a la derecha automáticamente, centrar títulos, llevar la cuenta de las páginas, poner notas al pie, encabezamientos, etc. Cabe decir que fue justamente con uno de estos procesadores como se escribió en su totalidad este libro. En el capítulo 4 se dedica una sección a los procesadores de palabras.

Sistemas de edición por computadora

Más aun, mediante las impresoras de láser es perfectamente factible que la computadora, con los programas adecuados, se encargue no sólo de la formación del texto, sino del diseño mismo de los caracteres que lo componen,

usando complejas técnicas que combinan el tradicional arte de la creación y el dibujo de letras con las funciones matemáticas que las describen rigurosamente. Los programas dedicados específicamente a la impresión de textos y fórmulas matemáticas, que poco a poco tomarán el lugar de los métodos tradicionales, han dado lugar a todo un nuevo campo llamado **tipografía matemática** (véase el artículo [BIGC83]). Uno de los principales autores de estos sistemas es Donald Knuth, quien además escribió una serie de textos que ya son clásicos de la literatura computacional (y que se mencionan más adelante en este texto). En 1986, en una entrevista, Knuth comentó lo siguiente:

> *Cuando vi que los caracteres [producidos por una computadora] lucían tan bien como los hechos con metal, me di cuenta de que habían sido creados sólo con unos y ceros. Yo no soy capaz de obtener nada con plomo, metalurgia ni nada de eso, pero con los ceros y unos la situación es distinta; creo que de eso sí entiendo algo. En realidad, se trata sólo de colocar ceros y unos en los lugares adecuados y entonces se puede tener una máquina que produzca los libros y se encargue de los problemas de la calidad**.

Otra de las capacidades de los equipos de cómputo actuales es la de representar la información de salida por medio de gráficas y dibujos. Las unidades especiales para estos fines reciben el nombre genérico de **graficadores**, y los hay de varios tipos, desde los muy sencillos hasta los altamente complejos y costosos.

Otras unidades de E/S

Cualquiera que haya dibujado una gráfica con una máquina de escribir sabrá que el problema principal que se presenta es el de la **resolución**; esto es, la capacidad de representar puntos discretos lo suficientemente cercanos entre sí para que aparenten continuidad. Uno de los parámetros principales para calificar un graficador, entonces, es la resolución.

Es posible convertir una pantalla de video en una terminal gráfica aumentando su resolución para que permita representar curvas y líneas a voluntad. Sólo es cuestión de escribir los programas adecuados para poder dibujar planos, mapas y figuras en tres dimensiones, de acuerdo con los principios establecidos en la geometría proyectiva (tarea que no es sencilla). De la misma forma es posible dibujarlos en papel, por medio de graficadores que mueven una o varias plumas sobre una hoja. El control del movimiento está, por supuesto, gobernado por un programa que la UCP ejecuta.

Los grandes graficadores (como los que dibujan los mapas topográficos) son computadoras especiales que reciben como entrada una cinta magnética que contiene millones de órdenes especiales, producidas por la computadora central, y que manejan cinco o más plumas de colores diferentes a enorme velocidad y resolución. Es tal la velocidad a la que dibujan, que la tinta tiene que ser bombeada hacia la pluma, ya que no llegaría a tiempo si simplemente fluyera por gravedad.

Por medio de programas especiales (llamados paquetes de graficación) es posible observar planos y diagramas complejos en una terminal de video, y

* Esta modesta opinión no deja siquiera suponer que Knuth ha publicado una colección de cinco volúmenes sobre tipografía digital, con lo que se ha colocado como uno de los creadores de este nuevo campo. Los sistemas son TEX y METAFONT, y la serie de libros, publicada por Addison-Wesley, se llama *Computers and Typesetting*.

moverlos, rotarlos, cambiarlos de escala y manipularlos a voluntad para manejar piezas de ingeniería mecánica o planos de arquitectura, por ejemplo, antes de que existan en la realidad. El campo que abarca todo esto, y que abre enormes posibilidades para el diseño gráfico, tipográfico y arquitectónico, recibe el nombre de **diseño auxiliado por computadora** o **CAD** (*Computer Aided Design*), y las terminales de graficación especiales para este fin se conocen como estaciones de trabajo.

En términos generales, es posible conectar virtualmente cualquier aparato a una computadora, para que funcione como unidad de entrada/salida. Es decir, una unidad de entrada puede ser, por ejemplo, un termómetro que controla cierto proceso que dependa de la temperatura. Siguiendo con este ejemplo, la unidad de salida puede ser un motor eléctrico que abre o cierra válvulas que logran el control del proceso en cuestión. En estos casos, el dispositivo está conectado a un **convertidor analógico/digital (A/D)**, que pasa la información analógica (cambios de temperatura) a información digitalizada (binaria), para que la computadora pueda procesarla. En la salida se requiere un **convertidor digital/analógico (D/A)**, que realiza la operación inversa.

En los últimos años han surgido sistemas de **síntesis de voz** en los que la salida de la computadora es en forma hablada, que se genera por medios electrónicos a partir de textos producidos por un programa. Es decir, en lugar de que la máquina imprima letras en una hoja de papel, una bocina emite sonidos que semejan la voz humana y que, con los avances tecnológicos, son cada vez mejores en cuanto a modulación y entonación. Es preciso aclarar que no se trata de voz humana, esto es, no está hecha de fragmentos pregrabados, sino que es el producto final de un complejo proceso electrónico de síntesis, basado en un enorme caudal de teoría matemática. A la entrada del convertidor digital/analógico llegan bits de información que son traducidos a fonemas y luego emitidos con su sonido correspondiente, con lo que es posible generar casi cualquier combinación de articulaciones. Se requiere, por supuesto, de un conjunto de programas para que la microcomputadora especial para este proceso pueda emitir palabras comprensibles a partir de textos.

El proceso inverso, que la entrada a una computadora sea por medio de voz humana que se traduzca a bits de información, es enormemente más complejo, y aun los sistemas más acabados son tan sólo capaces de reconocer un conjunto limitado de palabras (pronunciadas, además, por la misma persona y en forma pausada). Los obstáculos teóricos que hay que vencer para lograr la comunicación completa son de tal complejidad que se requiere de avances y descubrimientos sustanciales tanto en matemáticas como en las ciencias del lenguaje.

La tecnología digital también comienza a hacer su aparición en las técnicas musicales, y existe ya un conjunto de estándares para intercomunicación entre instrumentos musicales electrónicos (sintetizadores, percusiones electrónicas, etc.) llamado **MIDI** (*Musical Instrument Digital Interface*), que ya es de amplia utilización en las nuevas generaciones de dispositivos musicales, y que permite que una computadora controle el desempeño de un instrumento en forma automática, así como que grabe en un diskette información musical digitalizada.

Por otro lado, el **control de procesos en tiempo real** es otro gran campo de acción de las computadoras. Muchas máquinas-herramienta de reciente diseño integran uno o varios microprocesadores para que tomen decisiones al momento sobre el proceso que controlan. Supóngase, por ejemplo, una cortadora de rollos de papel controlada por un microprocesador. Tendrá un sensor óptico que servirá de unidad de entrada, y que supervisará la exactitud de los cortes. Cualquier desviación de la línea paralela que tiene registrada como patrón (representada, por ejemplo, por un haz de luz) será inmediatamente traducido por el convertidor A/D a señales digitales que el procesador analizará por medio de un programa. El resultado de esto será un conjunto de órdenes que hagan que la cuchilla se mueva algunas décimas de milímetro para mantener el corte deseado. Estas órdenes digitales deberán ser traducidas por un convertidor D/A para que muevan el servomotor que controla a la cortadora.

Como ésta, existen muchas nuevas aplicaciones de la computadora para controlar múltiples procesos. Lo que hasta hace poco se lograba por medio de complicados análisis matemáticos (parte de la llamada **teoría del control**), ahora se realiza con microprocesadores que vigilan que determinado proceso no se aparte demasiado de lo estipulado como función de salida. Y esto ocurre gracias a la retroalimentación que las unidades especiales de entrada/salida (y sus correspondientes convertidores) hacen posible.

En principio, cada vez que un dispositivo de entrada/salida intenta enviar a la memoria un byte (o recibirlo) ocurre, como se ha dicho, una interrupción: el procesador central abandona momentáneamente el proceso que estaba ejecutando y se dedica a atender al dispositivo que interrumpió, para luego proseguir con lo que estaba haciendo. Esto puede resultar inconveniente si el volumen de operaciones de entrada/salida es grande (y más aun cuando se trata de los dispositivos de memoria auxiliar que se describen a continuación), por lo que con el avance de la microelectrónica casi todas las computadoras actuales disponen de complejos circuitos que se encargan de la transferencia de datos entre los dispositivos periféricos y la memoria central, sin interrumpir constantemente al procesador central. Este método de acceso directo a la memoria (y los circuitos encargados de lograrlo) se conoce como **DMA** (*Direct Memory Access*). El libro [HAYJ79] es una buena referencia de éste y otros temas sobre arquitectura de computadoras.

Como último ejemplo de la gran variedad de unidades de entrada/salida que puede tener una computadora, se mencionará que en los grandes centros de cómputo se conecta una **microfilmadora** como unidad adicional de salida, para reproducir por medios fotográficos el gran volumen de información que normalmente aparecería impresa en papel.

3.4 Unidades de memoria auxiliar

Como la memoria central de una computadora es costosa y escasa, se vuelve necesario tener áreas adicionales de almacenamiento para guardar grandes

cantidades de información de manera más económica. Además, la memoria central pierde los datos almacenados al interrumpirse el suministro de corriente eléctrica, por lo que resulta poco práctico utilizarla para almacenamiento permanente de datos.

Estas y otras razones dan lugar a la creación de unidades periféricas de memoria que reciben, en conjunto, el nombre de **memoria auxiliar** o **secundaria**. Los medios físicos más comunes para almacenar información en estas unidades son las **cintas** y los **discos** magnéticos. El funcionamiento de estos aparatos es similar al de las cintas de audio (cassettes o cintas de carrete); esto es, los datos que se van a guardar en la cinta se representan mediante señales magnéticas que se reproducen y graban empleando una cabeza lectora/escritora.

La información residente en cualquiera de estos medios magnéticos recibe el nombre genérico de **archivo**. Un archivo está formado por un número variable de **registros**, generalmente de tamaño fijo, que pueden contener datos (numéricos o alfabéticos) o programas fuente escritos en algún lenguaje de programación. Los archivos que contienen programas, por lo común, son elaborados por el programador (digitados en una terminal de video), mientras que los de datos, sobre todo cuando son grandes, son introducidos a la computadora por mecanógrafos especializados llamados capturistas. Normalmente se hace referencia a los archivos empleando sus nombres simbólicos, asignados previamente por el programador.

Existen básicamente dos tipos de unidades periféricas magnéticas: unas en las que la información se lee/graba de manera **secuencial**, y otras donde el acceso a los datos es **directo** o **aleatorio**, es decir, sin importar el orden de lectura o escritura. El primer caso está representado por las cintas, y el segundo por los discos.

Almacenamiento secuencial

Esquemas de almacenamiento en cinta magnética

Las cintas magnéticas suelen manejarse en tres presentaciones: carrete, cassette, y cartucho magnético.

La información se almacena en una cinta magnética grabando cada byte (consistente en ocho bits) a lo ancho de la misma: los bits del 0 al 7 irán ocupando posiciones sobre una línea vertical hasta llenar todo el ancho de la cinta. De esta manera, los bytes se van acomodando uno por uno, a lo largo de la cinta magnética. El número de bits que caben a lo ancho determina el número de **pistas** (o canales) de la unidad de cintas. Actualmente casi todas las máquinas de carrete usan nueve pistas, aunque todavía se usan de siete. Si el número de pistas es menor que el número de bits en el byte (por ejemplo, para una cinta de siete pistas), se graban éstos divididos en grupos de cuatro o seis, con un bit —el séptimo— como separador. Generalmente, la última pista de la cinta se emplea para almacenar un bit de control —llamado de **paridad**—, que sirve como verificador de la consistencia de la información. Existen dos tipos de paridad: par e impar. En la paridad par, el último bit se escribe como 1 si es que en los ocho anteriores (para el caso de nueve pistas) existe un número non de unos, de modo que el número final

de bits en 1 sea par. En la paridad impar sucede lo contrario. Ambos tipos de paridad sirven para detectar la pérdida de información de un bit. Si una cinta, por ejemplo, está codificada con paridad par, será fácil detectar errores simplemente verificando que los canales tengan una cantidad par de bits en 1. Esta técnica también se usa en las memorias de semiconductores, para verificar que la información no se altere de manera accidental.

El número de bytes (o caracteres) que se pueden almacenar en una pulgada de cinta magnética determina la **densidad de grabación**, que se mide en **cpi** (caracteres por pulgada) o **bpi** (bits por pulgada), a lo ancho de la cinta. Un carácter por pulgada es equivalente a un bit por pulgada, como se comprenderá cuando se analice la siguiente figura, que contiene tres bytes codificados en nueve canales y con paridad par.

CANALES	CARACTERES...
0	1 1 0
1	1 0 0
2	0 1 0
3	1 0 0
4	0 0 1
5	1 1 1
6	1 0 1
7	0 0 1
P	1 1 0

Las densidades típicas de grabación para cintas de carrete son de 800 o 1600 bpi. Entonces, en una cinta de 2400 pies de longitud, a 1600 bpi, cabrían $1600 \times 2400 \times 12 = 46\,080\,000$ caracteres (12 porque hay ese número de pulgadas en un pie).

Lo anterior, sin embargo, no es correcto, por la siguiente consideración. Si la cinta se mueve a 45 pulgadas por segundo (que es una velocidad común), entonces pasarán bajo la cabeza lectora/grabadora $45 \times 1600 = 72\,000$ caracteres en un segundo. Por lo general, este número es mucho mayor que la cantidad que la unidad de cintas puede manipular cada vez, y será también mucho mayor que lo pedido por una sola operación de entrada/salida, por lo que hay que adecuar la cantidad de caracteres que se pueden leer en un segundo a las capacidades de almacenamiento en memoria de la UCP. Esto se logra dando **formato** a la cinta magnética; esto es, dividiéndola en

registros y bloques. Un registro* es la cantidad de bytes que la unidad de cintas puede leer/escribir en una sola operación; por lo común, es del orden de unos miles de bytes.

Pero ahora existe otro problema: debido a la inercia, es imposible que el motor de la unidad de cintas se detenga exactamente donde termina un registro y comienza el siguiente, por lo que hay que dejar un tramo de cinta sin grabar para evitar que el desplazamiento de la cinta (al frenarse el movimiento) haga que la cabeza lectora pierda su posición. Estos huecos en la cinta reciben en inglés el nombre de *inter-record gaps*, y comúnmente miden entre 0.5 y 1 pulgada. Está claro que, entonces, el número de caracteres que caben en una cinta se reduce casi a la mitad.

Por otra parte, cuando la información que se va a grabar en la cinta consiste en agrupamientos de pocos bytes (por ejemplo, nombres de personas con diez o doce letras) es necesario reunirla con la demás información del mismo tipo, para evitar desperdiciar (en *inter-record gaps*) casi toda la cinta. Estos agrupamientos de información (lógicos, no físicos) son los bloques en que se divide la cinta al darle formato. Así, cuando se dice que está bloqueada, se quiere decir que al leerla hay que separar, por medio de un programa, lo que previamente fue agrupado para no desperdiciar espacio. Este es un requerimiento que casi todas las computadoras resuelven mediante un conjunto de programas que forma parte del sistema operativo.

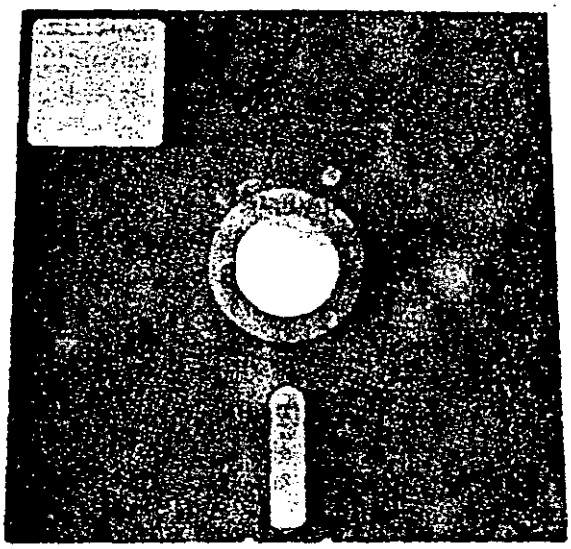
En lo que respecta a los cassettes y los cartuchos magnéticos, el tratamiento de la información es semejante, cambiando únicamente la cantidad de canales y las densidades de grabación y velocidades de movimiento de las cintas. En los cartuchos se manejan densidades de 6250 bpi, por lo que en un paquete de aproximadamente 15 × 8 × 2 cm caben varias decenas de millones de caracteres. No ocurre así con los cassettes, usados por las microcomputadoras de bajo costo y poca demanda de información, debido a que permiten una baja densidad de grabación, a poca velocidad.

La principal característica del almacenamiento secuencial consiste precisamente, en que sólo es posible leer, por ejemplo, el registro número 10 después de haber leído —e ignorado— los nueve primeros. Esto puede ser grave cuando se trata del registro número 15 000, porque hay que esperar a que la cinta se mueva varias decenas de metros, con una considerable pérdida de tiempo. Piénsese tan sólo que mientras la cinta da algunas pocas vueltas, el procesador puede ejecutar varios millones de instrucciones, por lo que es deseable disponer de métodos de acceso más rápidos.

Almacenamiento directo

Los discos son el medio que con mayor frecuencia se utiliza para esta forma de leer/grabar la información, y los hay en diversas presentaciones: discos

* Nótese que la palabra *registro*, en este caso, tiene un sentido distinto del empleado al hablar de la UCP o de los archivos. En el lenguaje de la computación, esta palabra tiene diferentes significados, dependiendo del contexto en que se emplee.



Diskette

Esquemas de almacenamiento en disco magnético

rígidos fijos, discos rígidos removibles y pequeños discos flexibles llamados **diskettes**. En términos generales, las unidades de acceso aleatorio son más costosas que las de acceso secuencial, pues los circuitos electrónicos requeridos para el movimiento de las cabezas lectoras/grabadoras son complejos y de gran precisión, como se verá a continuación. (Véase el artículo [WHIR80] para información más detallada.)

Un disco rígido consiste en uno o más platos o superficies magnéticas (casi siempre se emplean las dos caras de cada plato, con excepción del primero y último), montados junto con otros sobre un eje común. Para cada superficie existe una cabeza lectora/grabadora montada en un brazo que puede desplazarse en sentido radial, es decir, acercándose o alejándose del centro del disco, que gira constantemente a gran velocidad. En cada superficie, los datos se almacenan en pistas, organizadas como círculos concéntricos. Cada pista, a su vez, está dividida en porciones llamadas **sectores**. Visto por sectores, el funcionamiento de los discos es similar al de las cintas magnéticas, ya que en cada uno la información se almacena de manera secuencial. La diferencia consiste en que en el disco la cabeza sí puede ir directamente de una pista a otra (moviendo el brazo hacia o desde el centro) y, una vez en una pista, puede dejar pasar sectores (recuérdese que el disco gira constantemente) hasta que llegue al sector deseado.

Viendo el disco por arriba, todas las pistas de los diferentes platos que lo componen están alineadas (es decir, ocupan la misma posición en planos paralelos entre sí), y se conocen como **cilindros**. Un cilindro, entonces, es a un disco completo lo que una pista es a una de sus caras. Así, un disco que tenga 8 superficies con 1024 pistas cada una, por ejemplo, tendrá 1024 cilindros, aunque el número total de pistas será 8 × 1024.

Los parámetros para calificar una unidad de discos son la velocidad de rotación (y, por tanto, el tiempo que toma localizar una sección del disco, llamado **tiempo de latencia**) y el tiempo que el brazo tarda en moverse entre pista y pista (llamado **tiempo de acceso**, *seek-time* en inglés). Por lo común, los discos rígidos tienen más capacidad que las cintas, además de su muy superior velocidad. Para dar un ejemplo, una unidad sellada de disco rígido fijo de tecnología conocida como "Winchester"*, almacena 140 MB de datos, tiene cinco platos de 5 pulgadas de diámetro cada uno, gira a una velocidad de 3600 rpm, con un tiempo de latencia de 8 milisegundos (ms) y un tiempo de acceso de 28 ms en promedio para cada una de las 1024 pistas que incluye cada una de sus 8 superficies.

Cuando el disco magnético puede ser retirado de la unidad (y reemplazado por otro), se trata de discos removibles (*packs*), lo que permite formar verdaderas bibliotecas de ellos, como ocurre con las cintas. Los discos fijos presentan el problema de que si se dañan, la información contenida en ellos se pierde, por lo que en general suelen respaldarse periódicamente en varios carretes o cartuchos de cinta magnética. Esta operación de respaldo se conoce en inglés como *backup* o *dump*.

Los discos flexibles, por otro lado, son pequeños platos de material plástico que almacenan entre doscientos mil y un millón de caracteres, a velocidad relativamente baja y con poca densidad. Su ventaja está, por supuesto, en el precio, ya que cuestan mucho menos que los discos rígidos. Se usan sobre todo en micro y minicomputadoras. La principal diferencia operativa con respecto a los descritos líneas atrás es que en las unidades de diskettes, la cabeza lectora/grabadora está apoyada físicamente sobre la superficie del disco, mientras que en los otros "vuela" por encima del plato, a unas milésimas de pulgada. Esto significa que los discos flexibles se desgastan con el uso (igual que las cintas) mientras que los rígidos son virtualmente indestructibles bajo operación normal.

Ya sea por medio de discos o cintas, servir como almacenamiento de largo plazo no es la única función de la memoria secundaria. Desempeña un papel más interesante: servir de apoyo a la memoria central. Si se considera que, como se ha visto, el procesador sólo puede ejecutar instrucciones residentes en memoria central, entonces se puede pensar en llevar y traer a gran velocidad información entre los discos magnéticos y la memoria central, para aparentar ante el sistema que la memoria central es mucho más grande de lo que realmente es. Este esquema, que se estudia en el capítulo 4, se conoce como **memoria virtual.**

Una instalación típica pequeña, con un microprocesador de 16 bits, suele tener hasta 512 KB de memoria central, y unidades de diskettes para almacenar unos 500 KB en cada uno. Una máquina mediana usualmente tiene 1 MB o más de memoria central, y unidades de disco rígido que almacenan 80 o más megabytes. Una computadora grande suele tener 8 MB o más de memoria central, varias unidades de cinta magnética para almacenar de-

* Este curioso nombre viene de la configuración original que se pensaba dar a las primeras unidades comerciales que empleaban esta tecnología. Iban a consistir en una porción que almacenara 30 MB en la parte fija de la unidad, y 30 MB adicionales en una sección removible, es decir, 30-30.

cenas de megabytes en cada una, y discos removibles que guardan varios cientos de megabytes por unidad. Los grandes bancos de datos, llamados **bases de datos** o **sistemas de información**, suelen estar almacenados en bibliotecas de decenas (y a veces cientos) de discos removibles de gran capacidad.

Estos sistemas de programación "virtualizan" la información contenida en los archivos magnéticos, de tal manera que los usuarios puedan acceder a ella de manera simbólica, haciendo preguntas sobre el contenido más que sobre la configuración. En una base de datos es posible mantener, por ejemplo, la información concerniente a una biblioteca pública, para consultarla de la siguiente manera: se desea saber cuántas veces se ha prestado el libro X en los últimos dos meses, o se quiere saber cuántos libros hay sobre arqueología. El diseño de bases de datos y sistemas de información casi constituye una ciencia por sí mismo, por lo que se estudiará con más detalle en el capítulo 4.

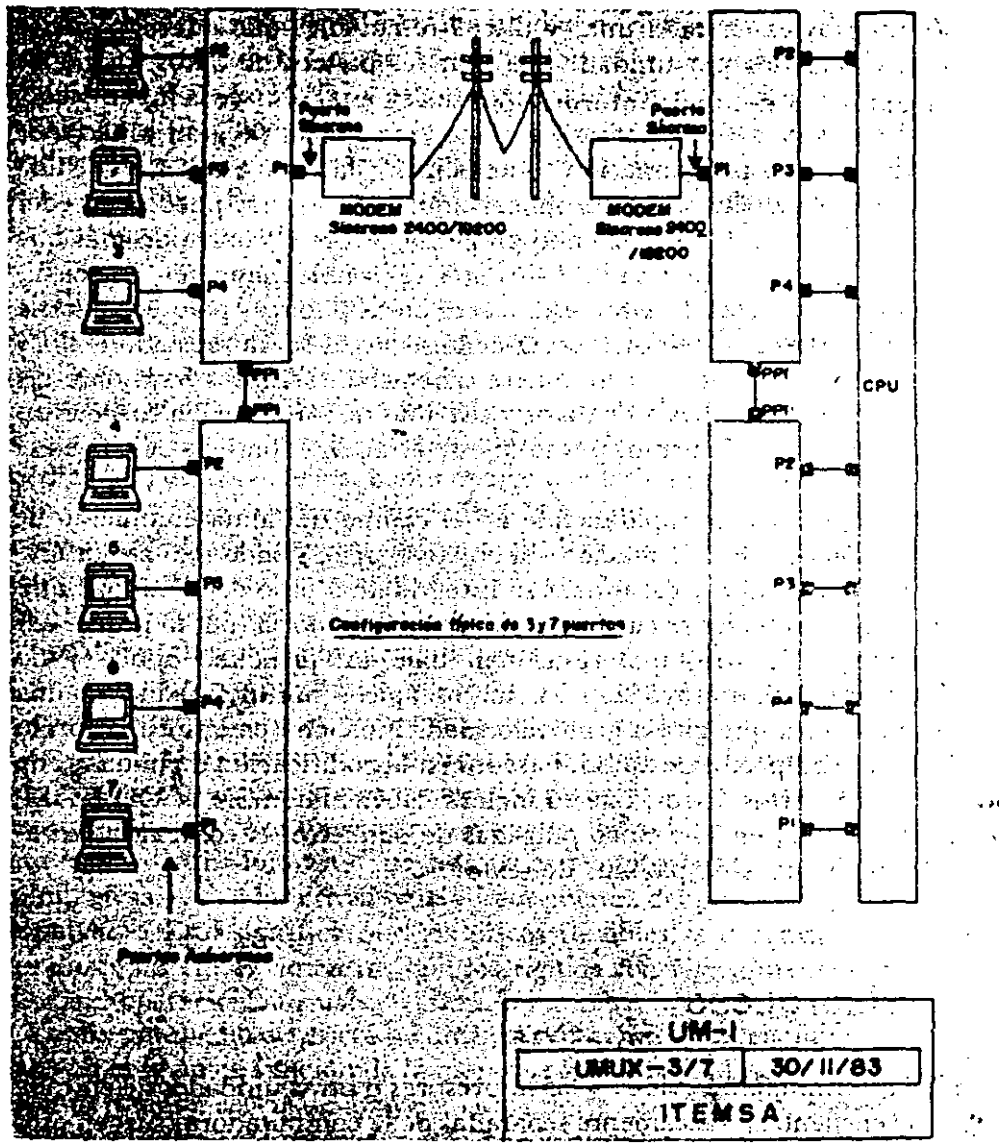
La tecnología avanza rápidamente en el campo del almacenamiento de grandes volúmenes de información. El objetivo sigue siendo reducir los costos por bit almacenado, y garantizar su integridad al paso del tiempo. Entre las posibilidades actuales se cuentan unidades de memoria (por lo pronto exclusivamente de lectura) que funcionan bajo un principio óptico y no magnético, en el que un rayo láser lee microscópicos puntos grabados en una superficie metálica que gira a gran velocidad. Funciona en la misma forma que los discos compactos de audio, basados en la codificación digitalizada de la información. Estos discos, que en inglés reciben el nombre de CD-ROM, pueden almacenar, en sólo cinco pulgadas de diámetro, el contenido de varias decenas de miles de páginas de texto.

3.5 Teleproceso

Cuando las terminales de video y las impresoras u otras unidades de entrada y salida se encuentran físicamente separadas de la computadora, surgen problemas que requieren un estudio aparte y que dan origen a una rama de las ciencias de la computación que se conoce como **teleproceso** o **teleinformática**

En efecto, nada obliga a la cercanía física (o geográfica en el caso extremo) entre la unidad central de procesamiento y los demás componentes del sistema de cómputo, puesto que todo lo que se requiere es un canal adecuado de comunicación para ligarlos. Este canal suele estar formado de cables en el caso de la computadora centralizada, pero también puede estar constituido por equipos complejos de telecomunicación.

Lo más sencillo sería conectar las terminales de la máquina por medio de cables muy largos, pero esto da lugar a ruidos e interferencias eléctricas en la línea. Entonces, hay que conectarlas por medios telefónicos o, si fuera el caso, de radiocomunicaciones. Para ambas situaciones se requiere una interfaz que tome los impulsos eléctricos digitales que manda la UCP y los convierta en señales que puedan transmitirse fácilmente a grandes distancias. Donde se reciben se requiere la operación inversa para volver a convertir esas



Ejemplo de una red de teleproceso

señales en las que espera la terminal. El aparato empleado para esto recibe el nombre genérico de **modem** (**mod**ulador/**dem**odulador), y es de uso común en todo aquel lugar donde sea necesario tener terminales remotas.

Unidades de medida de velocidades de transferencia

Hay dos unidades de medida que se emplean en los modems y en los equipos de comunicación digital. Una, llamada **bps**, (bits por segundo) mide la cantidad de bits que se transmiten en un segundo por el dispositivo o el canal, y es comúnmente empleada. Se habla, entonces, de modems de 1200, de 2400 o de 9600 bps, y para averiguar cuántas letras o símbolos se pueden transmitir por unidad de tiempo, hay que considerar que cada carácter (en código ASCII o EBCDIC) requiere 7 u 8 bits para ser codificado, además de algunos bits extra que se emplean regularmente para propósitos de control. Otra unidad, llamada **baud**, define la cantidad de transiciones lógicas

(entre los estados 0 y 1) que los circuitos digitales deben hacer para lograr la transmisión de un dato. En principio, cada bit transmitido cuenta por un baud, pero los modems suelen emplear esquemas especiales de modulación para reducir la cantidad de transiciones necesarias para enviar un bit, por lo que no siempre es el caso que un baud corresponde a un bps, sino que a veces con el costo de un baud es posible transmitir dos o más bits.

Para situaciones en donde se emplean varias terminales remotas distribuidas, son necesarios además equipos más complejos, que se encargan de concentrar varias señales y enviarlas por una línea telefónica común, o por medio del servicio de microondas. Aquí la situación se vuelve más complicada, porque ahora no sólo basta con modular y demodular la información, sino que también hay que entrar en redes públicas o internacionales, para lo que hay que cumplir con ciertos protocolos y estándares de comunicación*, y que pueden incluir la comunicación por medio de satélites. En tales situaciones es común emplear sistemas de **multiplexaje** y comunicación remota que deben ser atendidos por ingenieros especializados en comunicaciones, en lo que se conoce como **procesamiento digital de señales**.

En un sistema configurado de esta manera es posible que un usuario se encuentre a cientos (o miles) de kilómetros de distancia del lugar donde reside el procesador central y los discos magnéticos. Para tales casos es frecuente conectar, junto con las terminales remotas, una o varias impresoras a las que la UCP envía los resultados de los cómputos iniciados remotamente. Un sistema o subsistema de este tipo se conoce en inglés como **RJE** (*remote job entry*) y es, para todos los fines prácticos, una verdadera computadora local que depende de la instalación central.

Tal es el caso, por ejemplo, de las oficinas de reservaciones de aerolíneas, donde se agrupan varias terminales remotas, conectadas por radiotelefonía a la máquina central, que en ocasiones ni siquiera reside en el país donde está la oficina de boletos.

Y este hecho revela una de las direcciones que puede tomar la computación, y que consiste en descentralizar el procesamiento de datos (o regionalizarlo), y usar la computadora central únicamente para fines de consulta a sus enormes bancos de información, que residen en decenas de sus unidades de discos magnéticos. Se habla entonces de redes de computadoras o bien de procesamiento distribuido, que a continuación se explica.

Las **redes de computadoras** consisten en equipos de cómputo interconectados en forma directa o remota, de manera que comparten desde información almacenada en archivos hasta sus propios procesadores centrales. Las operaciones más usuales en una red son:

- transferencia o consulta de archivos de una máquina a otra,
- utilización de algún procesador existente en la red desde cualquier terminal,

* Se llama **protocolo de comunicaciones** al conjunto de normas técnicas que se deben cumplir para interconectar varios equipos. Como éste es un problema de alcance internacional cuando se trata de equipos que residen en distintos países, existen algunos organismos públicos que se han dedicado a la especificación de estándares. Entre los protocolos conocidos y de aplicación relativamente común se encuentran el X.25 y SNA (*Systems Network Architecture*, de IBM). La sección 3.7 es un anexo en el que se describe un modelo de normas internacionales que es respetado por esos y otros protocolos.

- terminal virtual,
- correo electrónico.

La primera operación permite enviar información de una computadora a otra cualquiera conectada a la red, ya sea para copiar archivos completos (de un disco a otro) o para usar los archivos de una como datos de entrada para un programa que se ejecuta desde otra. La segunda operación da la posibilidad de dirigir la ejecución de un proceso determinado a alguna UCP en particular dentro de la red, y la tercera hace aparecer una terminal de video como si estuviera conectada a cualquiera de las computadoras participantes, con la posibilidad de elegir la que se desee por medio de una simple orden. El concepto conocido como "correo electrónico" ofrece ese tipo de servicios entre los diferentes usuarios de la red, y asigna a cada uno un "buzón" en el que los demás pueden dejar depositados mensajes.

En una red, las computadoras pueden interconectarse con diversos métodos, que van desde un cable común hasta complejos sistemas de telecomunicaciones, e igualmente los datos pueden ser transmitidos con diversos grados de seguridad y complejidad. Uno de los proyectos que iniciaron, en la década de 1970, el campo de las grandes redes de computadoras fue el conocido como *Arpanet*, a cargo de la Oficina de Proyectos Avanzados del Departamento de Defensa de los Estados Unidos, que introdujo un método para enviar datos agrupándolos en "paquetes" individuales (y no mandándolos como un flujo ininterrumpido), conocido como *packet switching*. La ventaja de esto es que, como cada paquete está numerado e identificado, pueden ser transmitidos por diversos canales, así como ser reexpedidos si alguno llegara a perderse. Entre los sistemas de interconexión que se han popularizado, cabe resaltar uno llamado *Ethernet*, que tiene visos de convertirse en relativamente estándar porque ya ha sido adoptado por varios fabricantes. Consiste en un cable coaxial que se tiende en una oficina o un edificio, al que se conectan terminales, impresoras y computadoras mediante pequeños adaptadores. La información de la red (paquetes de datos y de control) fluye por el cable y permite las operaciones usuales de transferencia de archivos y de uso compartido de impresoras y discos magnéticos.

Existen grandes redes internacionales que ligan a muchas computadoras en varios países, y que se usan para labores especializadas, como en el caso de reservaciones de aerolíneas, o la red mundial de telecomunicaciones para finanzas, llamada *Swift*, que interconecta a más de 1 500 bancos en 39 países y permite el manejo de transacciones entre todos ellos.

Un esquema que ha surgido recientemente es el de **red local, LAN** (*local area network*), en donde un conjunto de computadoras personales se interconecta mediante un cable y puede (con la programación apropiada) compartir recursos tales como unidades de disco rígido o impresoras.

El concepto de **procesamiento distribuido**, por otro lado, dota a las terminales de lo que podría denominarse inteligencia local, pues son capaces de ejecutar partes del procesamiento *in situ*, sin tener que recurrir a la UCP más que para algunos casos especiales. Ejemplo de esto lo constituyen algunas terminales de video complejas que tienen memoria propia, en las que el usuario

puede hacer y rehacer textos completos sin depender de la computadora central, porque todo se almacena temporalmente en la propia terminal, que manda a la UCP los datos cuando ha terminado de preprocesarlos.

Muchos sistemas de información y captura de grandes volúmenes de datos (en los bancos, por ejemplo) tienen terminales especializadas que muestran al capturista una pantalla prediseñada, para que simplemente digite los datos pedidos sin preocuparse por la posición de cada uno de ellos. Estas terminales también pueden validar localmente los datos, para evitar errores de digitación. Una de estas máquinas puede servir, por ejemplo, para registrar los comprobantes de consumo de las tarjetas de crédito. En este caso aparece en la pantalla de video un diagrama para que el mecanógrafo especializado llene los huecos con la información que lee de los documentos que está captando. Si le corresponde, por ejemplo, escribir el monto numérico de la operación, la terminal le impedirá teclear cualquier cosa que no sean dígitos, evitando errores que pueden ser muy comunes. Todo este procesamiento tiene lugar sin intervención de la computadora central, que sólo recibirá los datos completos y depurados.

Con la llegada de las microcomputadoras se han abierto nuevas posibilidades para el procesamiento distribuido, que constará de grandes redes con inteligencia de cómputo, localizadas en diversos puntos, que toman información de su entorno, la preprocesan y la mandan, ya procesada, a la computadora central, para que pase a formar parte de los recursos del sistema de información.

Otro ejemplo de esto puede ser una red nacional ecológica (o sismológica o meteorológica) en la que múltiples estaciones dotadas de poder de cómputo propio analizan o muestrean los datos de la localidad donde están instaladas (cantidad de contaminantes por unidad de tiempo en un río, muestras de temperatura y presión cada minuto, lecturas de sismógrafos locales, etc.) y los hacen llegar —ya normalizados de acuerdo con criterios establecidos— a la máquina central para que ésta, a su vez, los integre como información ya clasificada.

El autor del presente texto participó en el diseño de una red de más de 600 microcomputadoras especializadas, que sirven de "capturadoras inteligentes" para exámenes escritos de opción múltiple. Cada uno de los participantes contesta su examen en la máquina; ésta preprocesa la información, cuidando que las respuestas sean todas permisibles y plausibles, y muestra, en menos de un segundo, la calificación obtenida. Luego se envían los datos a la computadora central para que ésta los integre y obtenga las listas de calificaciones y los resultados finales agrupados.

Esta descentralización también concierne al software, porque entonces se pide, por ejemplo, que las bases de datos estén distribuidas en múltiples máquinas, y que el sistema de información se encargue de dar una imagen de coherencia y unicidad, cuando en realidad los datos pueden estar almacenados en diferentes discos magnéticos que pueden estar separados por miles de kilómetros. Las **bases de datos distribuidas** representan un campo de investigación avanzada en ciencias de la computación.

La tendencia actual es descentralizar cada vez más el poder de cómputo, situando los sistemas en los lugares donde se requieren directamente, razón por la cual se debe esperar un aumento en la utilización de sistemas distribuidos o configurados en redes.

3.6 El sistema de cómputo integrado

No es válido llamar computadora a una máquina que no presente una imagen integrada y coherente a sus usuarios. Es decir, si hay que tener conocimientos de electrónica o de física para manejar un equipo, es que algo anda mal. Esto no quiere decir que cualquiera puede usar una computadora sin capacitación previa; significa más bien que el equipo de cómputo debe considerar la existencia de los usuarios y estar a sus órdenes de alguna manera no demasiado complicada. En efecto, esto presupone la existencia de interfaces entre el procesador, la memoria central y las unidades periféricas, por un lado, y quienes las manejan, por el otro, de modo que la comunicación sea lo más fluidas posible. Esto se convierte en un requisito indispensable cuando más se desea que la computadora se integre a las actividades usuales de la sociedad.

¿Qué es la informática?

Con la aparición de la computadora han surgido ramas de actividades afines, pero más amplias, que han recibido nombres como **procesamiento electrónico de datos** o **informática**. Este último concepto —aún amorfo— que abarca conocimientos y disciplinas de tipo matemático, computacional, administrativo y jurídico, se refiere a la utilización de la herramienta computacional para el desempeño de actividades de espectro o alcance mayor al de cualquiera de sus partes aisladas; su campo de acción cubre grupos sociales específicos (la empresa, la organización, la fábrica) y, en opinión de algunos, a la sociedad como un todo. Desde un punto de vista, la informática es el conjunto de técnicas necesarias para la creación de sistemas de información, que es tal vez el resultado final de todos estos esfuerzos. La referencia [LUCH76] ofrece un serio y accesible estudio sobre sistemas de información.

Creemos que la informática debe estar sustentada en los conceptos centrales de la ciencia de la computación y, sobre todo, en los conceptos matemáticos de modelo, algoritmo y sistema, que se estudian más adelante.

Un equipo integrado bajo estos principios presenta una imagen monolítica ante el usuario. Se describirá ahora una sesión típica de trabajo con una máquina de este tipo, suponiendo que se tiene acceso a una de sus terminales de video. La operación de una computadora personal es, por lo general, más sencilla de lo que se expone, pues aquí se hace referencia a un sistema que atiende a varios usuarios simultáneamente (conocidos como sistemas multiusuario).

Una sesión en un sistema multiusuario

Lo primero que el usuario potencial debe hacer es identificarse, mediante una clave asignada de antemano por el administrador del centro de cómputo. Entonces, la computadora busca, en un archivo especial en disco magnético, la clave recién digitada. Si la encuentra, permite la entrada al sistema y da

inicio la sesión; en otro caso, rechaza el intento empleando algún mensaje adecuado.

Ahora, el usuario puede consultar alguno de los archivos que ha creado con anterioridad, crear uno nuevo, ejecutar un programa o hacer alguna consulta a un banco de información, por ejemplo.

Estas acciones se comunican a la computadora por medio de un lenguaje especial, llamado lenguaje de control del sistema operativo. Dependiendo de la computadora de que se trate, este lenguaje puede ser muy sencillo o altamente complejo y elaborado.

Supóngase que se desea hacer un programa para calcular una tabla de amortizaciones e intereses bancarios. Lo primero que hay que hacer es resolver el problema en papel, escribiendo un programa en "pseudocódigo", para luego traducirlo a un lenguaje de programación particular. En este momento no nos preocupará cómo hacer esto, ya que la segunda parte de este texto trata precisamente de ello. Para este ejemplo se supondrá que el programa ya está escrito en una hoja de papel, y que lo único que hay que hacer es transmitirlo a la computadora.

Se logra esto mediante un programa especial del sistema que tiene como función servir de intermediario entre los usuarios y el sistema de archivos de la máquina. Este programa auxiliar se llama **editor de textos**. Por medio del editor, entonces, se escribe el programa y se deja en la computadora. (En el capítulo 4 se dedica una sección a los editores.)

Cuando se ha terminado de teclear el texto —y se han corregido los posibles errores mecanográficos también por medio del editor—, entonces hay que guardarlo; esto es, hay que convertirlo en un archivo en disco. La mayoría de los sistemas de cómputo tienen facilidades integradas en el lenguaje de control para asignar nombres simbólicos a los archivos. En el ejemplo este programa se llamará TABLA.

Los pasos realizados han sido:

1. Digitar la clave de usuario para entrar al sistema.
2. Llamar al editor.
3. Digitar el programa, y corregir los posibles errores tipográficos.

Ahora llega el momento de traducirlo al lenguaje propio de la máquina, para que el procesador pueda ejecutarlo. Para esto hay que llamar al traductor (que forma parte del sistema y reside en el disco), conocido como **compilador***.

Supóngase que el programa TABLA está escrito en el lenguaje COBOL. Así pues, el siguiente paso es

4. Llamar al compilador de COBOL para que compile el programa TABLA.

* Existe un compilador diferente para cada lenguaje de programación. Véase la sección 6.4 para la presentación de algunos lenguajes de programación.

Si la compilación no produce errores, se podrá ejecutar el programa para obtener las tablas de intereses. Es decir, el programa es general (está diseñado para producir cualquier tabla del tipo indicado); sólo hay que darle los datos para trabajar, que serán parámetros tales como el tipo de interés y los plazos, etc. La secuencia entonces prosigue:

5. Ejecutar el programa ya compilado y darle los datos que solicita.

El programa termina cuando ha finalizado el procesamiento de los datos, y el control vuelve al sistema de cómputo, que hará aparecer en la pantalla algún mensaje adecuado para indicar que se está de nuevo en un punto inicial, como el paso 2.

La sesión recién descrita ha sido *interactiva*, porque todos los procesos y pasos descritos han ocurrido en presencia del usuario y de la máquina, y con la participación de ambos. Esto significa que tanto los datos que pide la computadora como los resultados que entrega son manejados por un usuario desde su propia terminal de video, sin necesidad de que intervenga el operador del sistema. Una sesión interactiva permite, si el programa así lo especifica, hacer preguntas a la máquina y obtener las respuestas en el preciso momento en que se calculan, con las consecuentes ventajas.

Una vez que el programa está compilado y reside en el sistema de archivos de la computadora en forma de programa objeto, bastará con realizar los pasos 1 y 5 tantas veces como se desee para ejecutarlo de nuevo en el futuro.

Otra técnica —casi en desuso— para trabajar con una computadora, consiste en introducir los programas, datos y órdenes deseados por medio de tarjetas perforadas, para que el sistema de cómputo los procese en estricto orden secuencial y en un tiempo que no es el del usuario, sino que está determinado por las colas de servicio y la cantidad de proceso pendientes en el sistema. Esta técnica se conoce como **procesamiento por lotes** (*batch*).

El centro de cómputo

En vista de que una computadora grande es una herramienta de uso general, gran cantidad y diversidad de personas hacen uso de ella. El lugar donde reside la computadora, y a donde van a trabajar los usuarios, recibe tradicionalmente el nombre de centro de cómputo. No obstante, con la aparición de las microcomputadoras por un lado, y de nuevas técnicas de cómputo (procesamiento distribuido, teleinformática) por otro, ya no es estrictamente necesario que ambos, usuarios y computadora, estén en el mismo lugar.

Un centro de cómputo, por lo general, está dividido en áreas funcionales que se agrupan en dos familias: operativas, y administrativas y de apoyo. Las primeras incluyen, entre otras, las salas de máquinas, de impresoras y de terminales (o de perforadoras, si fuera el caso).

El lugar donde reside la UCP y las unidades de discos y cintas magnéticas es de acceso restringido y está controlado estrictamente por los operadores de la computadora. Estos operadores atienden los mensajes y pedidos que la UCP hace a través de una pantalla de video llamada consola; además, tienen

otras funciones importantes como realizar respaldos periódicos de todos los archivos del sistema a cintas o cartuchos magnéticos, y atender las solicitudes especiales de los usuarios.

Si en una de estas grandes máquinas alguien desea leer o escribir datos en una cinta, tiene que comunicárselo al operador central (por medio de su pantalla de video), para que éste coloque la cinta necesaria en la unidad, ya que las lectoras de cintas no están en la zona de terminales.

Cuando la sesión ha terminado, el usuario se dirige a la zona de impresoras para recoger sus resultados impresos, si así lo solicitó. Ahí lo atenderán otros operadores cuya tarea es recoger los listados que las impresoras producen para separarlos y acomodarlos en casilleros especiales, destinados a los usuarios del sistema de cómputo.

Las áreas administrativas y de apoyo de un centro de cómputo, por otro lado, incluyen la dirección y subdirección, una oficina de consultas y asesorías (donde los interesados pueden consultar manuales y resolver dudas), y oficinas especializadas de ingeniería y sistemas.

Toda máquina requiere atención y mantenimiento periódicos, por lo que los centros grandes de cómputo tienen uno o varios ingenieros residentes para estas funciones. Una sección de este departamento se dedica a mantener actualizados los inventarios de papel para impresión, que pueden ser de tamaño considerable (a veces se conoce a los gastos en papel como el "costo oculto" de un centro de cómputo, y llega incluso a rebasar los gastos normales de mantenimiento).

Igualmente se requiere el apoyo de ingenieros de software y de sistema operativo, que vigilan constantemente que los sistemas de programación de la computadora (compiladores, paquetes de programación, etc.) funcionen en forma adecuada y eficiente.

Es todo este grupo de personas (pueden ser decenas) el que provee el apoyo y la coordinación para que el usuario pueda llevar a cabo su trabajo en la gran computadora, para elaborar, corregir, probar o ejecutar programas, explotar bancos de información, o hacer casi cualquier cosa que su experiencia o imaginación dicten.

El usuario indica a la máquina que la sesión ha terminado por medio de una orden especial del lenguaje de control del sistema operativo, y la pantalla le informa el tiempo que estuvo en comunicación con el sistema de cómputo, y el tiempo que se utilizó la UCP durante la sesión, así como otros datos que pueden ser de interés. Luego, sin que el usuario lo vea, el sistema resta el tiempo de procesador empleado del total que éste tenga asignado (o comprado) en su cuenta personal, de tal manera que se lleva un control estricto de los recursos de cómputo empleados y su distribución.

Otros costos que el sistema contabiliza por cada usuario son el número de líneas impresas, la cantidad de espacio empleado en disco magnético, y el tiempo de utilización de las líneas de comunicación remota, si es el caso.

El centro de cómputo debe prestar atención individual a sus clientes al mismo tiempo que controla las tareas de todos en conjunto, preservando la privacidad de cada uno, y evitando en todo momento la sobrecarga de trabajo y la consecuente deficiencia en la atención. La magnitud y el alcance de

estas actividades son evidentes; considérese por ejemplo una gran computadora que da atención a decenas o cientos de personas a la vez, tiene cientos o miles de megabytes en memoria secundaria, maneja varias lectoras e impresoras, y procesa millones de instrucciones por segundo; entonces se puede reconocer la necesidad de una rama de las ciencias de la computación, especialmente dedicada a estos aspectos: la programación de sistemas, a la que se dedica el siguiente capítulo.

3.7 Anexo: estándar internacional para redes

En mayo de 1985 hubo una reunión de ejecutivos de las 20 compañías más grandes de computación y de comunicaciones para resolver el enorme problema de la falta de estándares para la comunicación entre sistemas. Como resultado de esa reunión se formó la Corporación para Sistemas Abiertos (COS, por sus siglas en inglés), que ahora agrupa a 60 instituciones y compañías, y que está trabajando activamente con los siguientes propósitos:

1. Seleccionar, de entre los estándares existentes, un conjunto que sirva como base.
2. Proponer y apoyar especificaciones para esos estándares.
3. Ofrecer una serie de pruebas para verificar el cumplimiento de esos estándares.

La idea es que en el futuro todo equipo de cómputo que cumpla con los estándares propuestos para la intercomunicación lleve el sello COS, poniendo fin al caos actual.

El estándar al que se llegará en un futuro cercano se llamará Modelo de Referencia para la Conexión entre Sistemas Abiertos (OSI, por sus siglas en inglés), y estará considerado por la Organización Internacional de Estandarización (ISO). Este organismo, fundado en 1947, agrupa en la actualidad a casi 90 países.

Estándar internacional para teleproceso

En 1978, la organización ISO propuso el llamado modelo de referencia de siete niveles, que define y gobierna la arquitectura estándar para la comunicación de información entre diversos equipos. Este estándar se terminó en 1984, cuando el Comité Consultivo Internacional de Telefonía y Telegrafía (CCITT) lo adoptó en forma conjunta con ISO como guía para la creación de redes internacionales de datos. Es decir, la agrupación industrial COS adoptó el modelo ISO como base para establecer estándares para la comunicación abierta entre sus diferentes sistemas, dentro del nuevo modelo de interconexión OSI, que funciona estructurado en los siete niveles jerárquicos que a continuación se describen.

Modelo jerárquico ISO/OSI

NIVEL 1 La capa física es la responsable de la conexión física entre el dispositivo y la red. Es aquí donde se definen detalles acerca de los conectores y las señales eléctricas.

NIVEL 2 La capa de comunicación de datos se encarga de la detección y correcccción de errores y de la organización de la información en paquetes que pueden ser contados y reexpedidos en caso de necesidad.
NIVEL 3 La capa de red es la responsable de asignar ruta a los paquetes de datos hacia su destino, y de escoger las mejores vías para ello.
NIVEL 4 La capa de transporte controla la integridad del mensaje desde su origen hasta su destino final, para lo cual puede alterar las prioridades y velocidades de envío y recepción de información.
NIVEL 5 La capa de sesión tiene como función controlar el flujo de información entre origen y destino, de modo que la sesión de comunicación no se altere y se mantenga el diálogo sincronizado entre origen y destino.
NIVEL 6 La capa de presentación traduce los datos del formato y lenguaje que se maneja en el nivel superior, al formato y lenguaje que se requiere para su envío y control; es decir, es la encargada de "esconder" al usuario los detalles de manejo de protocolos en la red.
NIVEL 7 La capa de aplicación maneja y presenta los formatos requeridos por la estación de trabajo o terminal para mostrar al usuario la información ya configurada, legible y óptima, de acuerdo con sus intereses particulares.

Así pues, se espera que, en el futuro buena parte de los equipos de cómputo puedan intercomunicarse entre sí en forma directa, ya que existe un consenso entre los principales fabricantes y usuarios para este fin.

Pronto se estará hablando de otras siglas adicionales, **ISDN**: Red Digital de Servicios Integrados, que manejará todas estas interfaces en una forma sencilla y accesible a los usuarios.

Palabras y conceptos clave

En esta sección se agrupan las palabras y conceptos de importancia que se estudiaron en el capítulo, con el objetivo de que el lector pueda hacer una autoevaluación que consiste en ser capaz de describir con cierta precisión lo que cada término significa, y no sentirse satisfecho hasta haberlo logrado. Se muestran en el orden en que se describieron o se mencionaron.

SISTEMA DE CÓMPUTO
PROCESADOR CENTRAL
INTERRUPCIONES
MIPS
FLOPS
REGISTRO DE LA UCP
ARQUITECTURA RISC
TAMAÑO DE PALABRA
MICROPROGRAMACIÓN
DETECCIÓN DE PARIDAD
VLSI
MEMORIA CENTRAL
CORE
RAM
ROM
PROM
FIRMWARE
PAL
MEMORIA DE BURBUJAS
TERMINAL DE VIDEO
CURSOR
LECTORA DE TARJETAS
PERFORADORA
IMPRESORA DE MATRIZ
IMPRESORA DE CADENA
IMPRESORA DE LÁSER
PROCESADOR DE PALABRAS
TIPOGRAFÍA MATEMÁTICA
GRAFICADOR
CAD/CAM
CONVERTIDOR A/D-D/A
SÍNTESIS DE VOZ
MIDI
CONTROL DE PROCESOS
DMA
MEMORIA SECUNDARIA
CINTA MAGNÉTICA
DISCO MAGNÉTICO
ARCHIVO
REGISTROS DE UN ARCHIVO
ACCESO SECUENCIAL
ACCESO DIRECTO

PISTA	CD-ROM	*PACKET SWITCHING*
PARIDAD PAR/IMPAR	TELEPROCESO	*ETHERNET*
DENSIDAD DE GRABACIÓN	MODEM	RED LOCAL (LAN)
BPI	BPS	PROCESAMIENTO DISTRIBUIDO
FORMATO DE UNA CINTA O DISCO	BAUD	INFORMÁTICA
REGISTROS Y BLOQUES	PROTOCOLO	CENTRO DE CÓMPUTO
DISKETTE	X.25	PROCESAMIENTO POR LOTES (*BATCH*)
SECTORES	SNA	COS
TIEMPO DE LATENCIA	RJE	MODELO ISO/OSI
TIEMPO DE ACCESO (*SEEK-TIME*)	RED DE COMPUTADORAS	ISDN
CILINDRO	TERMINAL VIRTUAL	
	CORREO ELECTRÓNICO	
	ARPANET	

Ejercicios

1. Obtenga los siguientes datos de alguna computadora multiusuario y haga una descripción comparativa contra lo mencionado en el texto:

   Velocidad del procesador central
   Número de registros que contiene el procesador
   Esquemas de microprogramación que emplea
   Cantidad y tipo de interrupciones que maneja el procesador
   Tamaño de la memoria central
   Velocidad de acceso a la memoria central
   Dispositivos especiales de entrada/salida
   Tipo de los discos magnéticos
   Capacidad y velocidad de acceso de los discos magnéticos
   Tipo de las unidades de cinta
   Facilidades de teleproceso
   Parámetros que el sistema contabiliza para cada usuario

2. Un concepto novedoso, que consiste en que un área grande de memoria RAM simula las funciones de un disco magnético, recibe en inglés el nombre *RAM-disk*. Investigue las características de alguno de ellos y describa sus ventajas y desventajas.

3. Un entusiasta lector de enciclopedias tiene la idea de comprarse una microcomputadora para almacenar ahí su *Encyclopaedia Britannica*. La computadora en cuestión es el modelo AT de las llamadas computadoras personales, que tiene unidades de almacenamiento en diskette, con capacidad de 1.2 MB cada uno. Ahora, considere los siguientes datos.

- La enciclopedia consta de 30 volúmenes, organizados de la siguiente forma:

  19 volúmenes de aproximadamente 1100 páginas cada uno; cada página está dividida en dos columnas de 80 líneas, con un promedio de 9 palabras por línea.

11 volúmenes de aproximadamente 1000 páginas cada uno; cada página está dividida en tres columnas de 90 renglones, con un promedio de 6 palabras por renglón.

- La longitud promedio de una palabra es de 5 letras.

Calcule el número de diskettes que se requieren para almace. .iciclopedia completa, considerando que se requiere un byte para representar una letra. (Los espacios en blanco también cuentan.)

4. Entre los problemas que enfrenta nuestro aguerrido lector y usuario de computadoras del problema anterior, el de la captura del texto que compone las miles y miles de páginas de la enciclopedia no es ciertamente el menor. Está claro que es prácticamente imposible teclear el contenido de los libros, así que sería muy interesante buscar formas más eficientes de introducir texto en una computadora. Existen unas unidades de entrada que son capaces de "leer" directamente texto impreso, y que evitan la tarea de reteclearlo. Averigüe su principio de funcionamiento y sus limitaciones.

5. El banco "El Seguro de su Ilusión" ha decidido controlar mediante computadora el manejo de las cuentas de cheques de sus 75 000 clientes, para lo cual se tiene que evaluar la cantidad de almacenamiento en disco rígido que debe adquirirse. Para calcular esta cantidad, primero hay que diseñar lo que en procesamiento de datos se conoce como el *registro* para cada cliente, que consistirá en un conjunto de *campos* para almacenar los datos requeridos, por ejemplo, de la siguiente forma:

Campo	Longitud
Nombre del cliente	30 caracteres
Dirección	50 caracteres
Teléfono	10 caracteres
Número de cuenta	15 caracteres
Fecha	6 caracteres
Monto del depósito inicial	18 caracteres
Descripción de una transacción	50 caracteres
Monto de una transacción	18 caracteres

Se estima que un cliente común hará entre 20 y 50 transacciones mensualmente; y para propósitos legales se requiere tener almacenados los movimientos de los últimos dos meses.

Investigue la cantidad de almacenamiento en disco que se requiere para enviar a cada cliente un informe mensual de su actividad bancaria, y determine otros campos que habrían de considerarse en el diseño del registro.

6. ¿Por qué es más complejo lograr que una computadora interprete mensajes hablados que lograr que los produzca?

7. Averigüe dónde está la computadora que se encarga de procesar las reservaciones de alguna aerolínea que tenga oficinas en su localidad.

Referencias para el capítulo 3*

[ATKT79] Atkins, Travis, "What is an interrupt?", en *Byte*, marzo, 1979.
Artículo para principiantes que explica, por medio de analogías y ejemplos, el concepto y función de las interrupciones en un (micro)procesador.

[BIGC83] Bigelow, Charles y Donald Day, "Digital Typography", en *Scientific American*, agosto, 1983.
Artículo que muestra los avances de la creación e impresión de textos por computadora. Menciona aspectos de la creación de los caracteres y de su almacenamiento en la memoria de la máquina, así como las facilidades disponibles para su manipulación. Existe traducción al español.

[CLIB79] Cline, Ben, "An Introduction to Microprogramming", en *Byte*, abril, 1979.
Aunque trata un tema avanzado, este artículo logra dar una idea adecuada en un nivel que sigue siendo intermedio. Existen, por supuesto, libros completos sobre este tema, así que no se puede esperar más que una explicación somera.

[CRAW86] Cramer, William, y Gerry Kane, *68000 Microprocessor Handbook*, 2a. ed., Osborne, McGraw-Hill, Berkeley, 1986.
Este es uno de varios libros acerca de los sistemas de cómputo configurados alrededor de los microprocesadores 68000, 68010 y 68020. Es un libro técnico, como todos los de su estilo.

[HALI79] Halsema, A. I., "Bubble Memories", en *Byte*, junio, 1979.
Interesante revisión, de nivel introductorio, de las nuevas tecnologías de almacenamiento en circuitos de alta integración.

* Estos artículos de revistas, cortos y accesibles, ilustran algunos de los conceptos descritos en el capítulo con un poco más de detalle, pero estas referencias son simplemente una guía. Si al lector le interesa estudiar estos temas más profundamente, se recomienda revisar la bibliografía estándar de computación que se cita en otros capítulos del libro.

La revista *Scientific American* publica ocasionalmente libros en los que reúne artículos agrupados por temas de interés; en su mayoría están traducidos, y los números mensuales de esta revista en español se publican (con un atraso de varios meses) con el nombre de *Investigación y Ciencia*.

La revista *Byte* (*"the small systems journal"*) se especializa en temas relacionados con microcomputadoras y microprocesadores, y suele contener artículos de nivel introductorio e intermedio de buena calidad.

[HAYJ79] Hayes, John, *Computer Architecture and Organization*, McGraw Hill International, Tokio, 1979.
Libro de nivel intermedio sobre arquitectura de computadoras. Tiene un capítulo completo sobre la evolución histórica de las generaciones de computadoras, considerándolas desde el punto de vista de su diseño. Explica el funcionamiento y las técnicas generales de diseño del procesador, la unidad de control, el sistema de memoria y la computadora como un todo.

[INTC87] Intel Corporation, *Microprocessor and Peripheral Handbook, Volume I: Microprocessor*, Intel Corporation, California, 1987.
Manual equivalente al del 68000, dedicado a la familia de microprocesadores 8088, 8086, 80286 y 80386 y sus aplicaciones. La industria de la computación avanza con tal rapidez que las compañías se ven obligadas a producir periódicamente manuales de este tipo.

[KUHL79] Kuhn, Larry y Robert Myers, "Ink-jet printing", en *Scientific American*, abril, 1979.
Artículo de interés general y de nivel introductorio (como todos los de esta revista) sobre las enormes impresoras llamadas *page printers*. Existe traducción al español.

[LUCH76] Lucas, Henry, Jr., *The analysis, Design and Implementation of Information Systems*, McGraw-Hill International, Tokio, 1976.
Interesante y conciso libro de conceptos básicos sobre estudio y creación de sistemas de información. A diferencia de otros sobre la materia, éste no incurre en generalidades fáciles ni en las muchas ambigüedades que desafortunadamente abundan en este campo.

[NEWW81] Newman, William y Robert Sproull, *Principles of Interactive Computer Graphics*, McGraw-Hill International, Tokio, 1981.
Segunda edición de un importante texto sobre diseño y creación de gráficas por computadora. Es un libro avanzado, que supone en el lector conocimientos de matemáticas y estructuras de datos. Tiene varios apéndices sobre técnicas matemáticas y otro sobre más de 500 referencias bibliográficas.

[PATD83] Patterson, David, "Microprogramming", en *Scientific American*, marzo, 1983.
Explicación introductoria sobre la microprogramación en donde se muestran algunos ejemplos de la teoría del funcionamiento de este esquema, básico para la operación de las computadoras grandes. Aunque trata un tema avanzado, el artículo expone de manera accesible la idea adecuada de la importancia del "microcódigo".

[SIED82] Siewiorek, Daniel, Gordon Bell y Allen Newell, *Computer Structures: Principles and Examples*, McGraw-Hill International, Tokio, 1982.
Nueva versión de un libro de nivel avanzado, considerado como la "biblia" de los textos sobre arquitectura de computadoras, originalmente publicado en 1971. Se integró un nuevo coautor y se expandió aun más el material cubierto: a lo largo de 900 páginas se describen con mucho detalle estructural más de 30 sistemas diferentes, desde micro hasta supercomputadoras. Se emplean dos lenguajes especializados en la descripción de arquitecturas de cómputo, llamados ISPS y PMS, usados para mostrar el funcionamiento de la unidad central de procesamiento y el lenguaje de máquina, en el primer caso, y para un nivel de abstracción más alto, en el segundo.

[VACA75] Vacroux, André, "Microcomputers", en *Scientific American*, mayo, 1975.
Muy interesante y completa descripción de las entonces recién surgidas microcomputadoras. Incluye fotografías de circuitos integrados a gran escala que ilustran las ideas del autor acerca de esta revolución tecnológica.

[WHIR80] White, Robert, "Disk Storage Technology", en *Scientific American*, agosto, 1980.
Artículo que explica el funcionamiento de los discos rígidos y sus aplicaciones, aunque desde un punto de vista más bien técnico. Existe traducción al español.

4

La programación de sistemas

Por **programación de sistemas** se entiende el conjunto de programas necesario para que una computadora dé una imagen coherente y monolítica ante sus usuarios. Se ha visto ya que una máquina tan rápida como las descritas sólo es capaz de hacer un pequeño número de operaciones muy elementales, y surge entonces la pregunta ¿cómo hacer para que el trabajo con una computadora sea eficiente y no haya que comunicarle todo por medio de ceros y unos? La respuesta a esta pregunta constituye, precisamente, el desarrollo de la programación de sistemas, a la que se dedica este capítulo.

4.1 Lenguaje de máquina

Todo lo que se ha descrito en el capítulo 3 ha estado en el lenguaje propio de la computadora, esto es, en un lenguaje que hace referencia a los registros de la UCP, al acumulador, a las celdas de la memoria, etc. Sin embargo, estos elementos de la arquitectura de la máquina están bastante lejanos del lenguaje cotidiano, por lo que ahora se comenzará a explorar las posibilidades de acercar la computadora al dominio del ser humano.

El siguiente es un programa (objeto) real, escrito en el código hexadecimal* del microprocesador Intel 8086, y sirve para encontrar un número entre un conjunto de números enteros mediante el método llamado de búsqueda lineal. Juzgue el lector la "claridad" del mismo:

```
1E B80000 50 B82810 8ED8 8EC0 BF0000 BB1D00 8B0F BB1F00 8A07
FC F2 AE 7401 CB 4F CB
```

Cada grupo de números representa el código —en el lenguaje de esta máquina— de una instrucción del procesador. Obsérvese que, de acuerdo

* Se llama hexadecimal al sistema de numeración con base 16.

con lo dicho en la página 34, a veces las instrucciones medirán un byte (dos dígitos hexadecimales), o dos (cuatro dígitos), o tres.

Los procesadores permiten varias maneras de "direccionar" (llegar a) las celdas de la memoria. La más simple de todas es el **direccionamiento directo**, que es la que se ha usado en los ejemplos; basta con escribir la dirección de la celda deseada a continuación del código de la instrucción para que la unidad de control logre el acceso.

En otra forma, el **direccionamiento inmediato**, se usa un dato numérico que se escribe inmediatamente a la derecha de la instrucción que la requiere. La diferencia entre ambos tipos de operaciones consiste en que la primera usa la dirección donde está el dato, mientras que la segunda usa el dato mismo. Por ejemplo, `CARGA_Ac 22` mete al acumulador el valor contenido en la celda con dirección 22, mientras que `CARGA_Ac 22(Inm)` mete el número 22 en el acumulador.

El **direccionamiento indirecto**, por otra parte, no toma el número que está a la derecha de la instrucción como dirección para extraer de ella un valor, ni como dato inmediato, sino como dirección de una celda a la que tendrá que ir para extraer otra dirección. Esto puede parecer rebuscado, pero es de enorme utilidad en la programación de sistemas.

Por último, el **direccionamiento indizado** usa la dirección que está a la derecha de la instrucción para sumarla con el contenido de un registro especial de la UCP llamado **registro índice**. Esto permite la variación de direcciones para simular directamente el recorrido sobre los elementos de un vector.

El 8086 puede direccionar memoria en todas las formas explicadas, y dispone de 25 tipos de combinaciones de estos métodos básicos. Además, maneja palabras de memoria de 16 bits (que para nuestros fines se pueden considerar como la unión de dos celdas de 8 bits).

Como se ha dicho, lo que este microprocesador puede hacer es poco, y se reduce a unas 130 operaciones primitivas diferentes —divididas en seis grupos— que a continuación se reseñan brevemente para dar al lector una idea de las capacidades del lenguaje de máquina (véase [MORC84]).

GRUPO DE TRANSFERENCIA DE DATOS

- Copia el contenido de un registro a otro (un número de 16 bits).
- Copia el contenido de una celda de memoria a un registro y viceversa.
- Copia un número (inmediato) a un registro o a una palabra de memoria.
- Intercambia los contenidos de dos registros o de un registro y una palabra de memoria.
- Intercambia los contenidos del acumulador con un registro o con una palabra de memoria.
- Guarda o extrae un número de 16 bits en la pila de control.
- Transfiere un número de 8 ó 16 bits entre el procesador y un dispositivo de E/S.

- Maneja el acceso a una tabla definida en la memoria.
- Carga direcciones de palabras de memoria.

GRUPO ARITMÉTICO

- Vuelve negativo un número de 8 o de 16 bits.
- Suma dos números enteros de 8 o de 16 bits (en varios modos de direccionamiento).
- Resta, similar a la suma.
- Multiplica dos números enteros de 8 o de 16 bits.
- Divide dos números enteros de 8 o de 16 bits.
- Incrementa en uno el valor de un registro o de una palabra de memoria.
- Decrementa, similar al incremento.
- Convierte un número de 8 bits en uno de 16, o uno de 16 en uno de 32.
- Representa el contenido de un registro de 8 bits como dato binario o como dato ASCII, para ajustarlo para la suma, resta, multiplicación y división.

GRUPO LÓGICO Y DE DESPLAZAMIENTOS

- Efectúa la operación **NOT** sobre el contenido de un byte o de una palabra.
- Efectúa la operación **AND** sobre el contenido de un byte o de una palabra.
- Efectúa la operación **OR** sobre el contenido de un byte o de una palabra.
- Efectúa la operación **XOR** sobre el contenido de un byte o de una palabra.
- Desplaza los bits de un registro o de una celda de memoria una o varias posiciones a la izquierda.
- Desplaza los bits de un registro o de una celda de memoria una o varias posiciones a la derecha.
- Rota los bits de un registro o de una celda de memoria una o varias posiciones a la izquierda, en diversas modalidades (con/sin acarreo, etc.).
- Rota los bits de un registro o de una celda de memoria una o varias posiciones a la derecha, en diversas modalidades (con/sin acarreo, etc.).

GRUPO DE MANEJO DE CADENAS

- Controla la transferencia de un bloque de caracteres.
- Mueve los caracteres de una cadena, considerados como bytes o como palabras.

- Compara los caracteres de una cadena, considerados como bytes o como palabras.
- Busca caracteres en una cadena, considerados como bytes o como palabras.
- Introduce caracteres en una cadena, considerados como bytes o como palabras.
- Almacena caracteres de una cadena, considerados como bytes o como palabras.
- Manejo del apuntador a una cadena.

GRUPO DE FLUJO DE CONTROL

- Salto directo a una cierta dirección (en varias modalidades).
- Salto condicional a una cierta dirección, dependiendo del resultado de una comparación anterior.
- Compara el valor de dos registros de 16 bits, y detecta si son iguales, o si uno es mayor o menor que el otro.
- Igual que el punto anterior pero para bytes.
- Igual que el punto anterior pero para un registro y un dato inmediato.
- Detecta si el contenido de un registro es cero, positivo o varias posibilidades más.
- Repite la ejecución de un ciclo predefinido.
- Repite condicionalmente la ejecución de un ciclo predefinido.
- Llama a una subrutina, con varias modalidades que dependen del tipo de direccionamiento.
- Regresa de una subrutina, con varias modalidades que dependen del tipo de direccionamiento.

GRUPO DE CONTROL DEL SISTEMA

- Provoca una interrupción.
- Regresa de una interrupción.
- Borra/activa el indicador de interrupción.
- Espera una señal de sincronización.
- Deshabilita el acceso al canal de datos del procesador.
- Guarda el estado actual de la UCP en la pila de control.
- Extrae el estado actual de la UCP de la pila de control.
- Operación nula (esto es, "no hagas nada" durante el ciclo).
- Alto

Aunque no esté claro para qué sirven todas estas operaciones, sí es posible darse cuenta de que no son muy potentes ni impresionantes, y que constituye un gran esfuerzo lograr hacer algo complicado con ayuda de tan escasos medios.

Hay que tomar en cuenta que la descripción que se acaba de hacer de este diccionario no entró en detalles, y que para completarla haría falta definir

los códigos binarios (o hexadecimales) correspondientes a cada instrucción, pero esto no se hará ya que no nos es de utilidad.

Un programa escrito en lenguaje de máquina tiene las siguientes características: está escrito en ceros y unos (el ejemplo está en hexadecimal, por claridad); hace referencias a celdas absolutas de la memoria y es inflexible, en el sentido de que no admite cambios para adaptarlo a nuevos requerimientos. Todo programa escrito en lenguaje de máquina deberá tener estas características, que lo vuelven totalmente ilegible e impráctico para todo fin humano (por ejemplo, mejorarlo, corregirlo, comentarlo, comunicarlo a otros, etc.). Esto implica que ya no se trabaja en lenguaje de máquina, puesto que ésta es la manera más inadecuada de comunicarse con una computadora. Lo que hay que hacer es encontrar un mejor medio. La primera idea que surge es usar el diccionario definido en la página 34 y ordenar a la computadora `CARGA_AC`, en lugar de `21`, con las correspondientes ventajas en cuanto a legibilidad y claridad en el concepto. Esta idea se estudiará con más detalle en la siguiente sección.

Características de un programa escrito en lenguaje de máquina

4.2 Ensambladores

Si se decide usar nuestro diccionario para comunicarse con la computadora, antes hay que resolver el problema de traducir el programa fuente (escrito con los mnemónicos del diccionario) a lenguaje de máquina (el único que la UCP admite).

Se podría pensar en contratar a un traductor (que llamaremos T1) para que a) lea cada programa fuente y b) lo traduzca a lenguaje de máquina. Si existe tal traductor, ya no es necesario trabajar en lenguaje de máquina, sino que se programaría en un lenguaje de más alto nivel. Claro que para que esto siempre funcione habría que ser capaces de integrar ese traductor a la máquina misma; en pocas palabras, que la máquina traduzca por sí sola los programas fuente a programas objeto.

¿Cómo se logra esto?

Si se escribe el programa traductor y se deja residente en la memoria de la computadora, el proceso de comunicación con ella tendría dos pasos: primero, convertir el programa fuente del usuario a programa objeto, y segundo, cargar y ejecutar ese programa objeto, que ya quedó escrito en lenguaje de máquina.

Ahora la comunicación con la computadora requiere dos pasos

Se analizarán los pasos necesarios para construir un traductor de esta clase. Se propone una manera sencilla de atacar problemas complejos de este tipo, que consiste en describir en español, a grandes rasgos, una solución general. Tal solución será un primer acercamiento al problema. Tal vez sean necesarios varios acercamientos progresivos para entender y resolver un problema complejo pero, por lo menos, ya se ha descrito una forma general de lograrlo; más que esto, se ha definido una metodología de diseño, que luego se explicará con detalle. Ahora, volvamos a nuestro problema.

Lo que se desea es hacer un programa T1 que reciba como entrada un programa fuente escrito en mnemónicos (usando el diccionario antes men-

cionado) y que produzca como salida el mismo programa, ya convertido a lenguaje de máquina, es decir, el programa objeto listo para ser ejecutado.

Diseño inicial de un primer traductor

Un primer acercamiento podría ser el siguiente:

```
! Programa "T1", primera versión.
! (El símbolo "!" se usa para escribir comentarios.)
Para cada renglón del programa fuente se ejecuta lo siguiente:
Buscar la palabra mnemónica en el diccionario.
Si está, entonces traducirla a lenguaje de máquina
(simplemente, leer la columna de la derecha de esa entrada en el diccionario).
En caso contrario mandar un mensaje de error que diga, por ejemplo,
"mnemónico desconocido".
```

Está claro que, en este nivel de detalle, el programa T1 funciona. Pruebe el lector aplicarlo al ejemplo de la página 35 y verá cómo pasa del programa fuente

```
CARGA_Ac    21
SUMA        22
GUARDA_Ac   23
ALTO
```

al programa objeto equivalente

```
21215722962370
```

suponiendo que T1 tiene acceso al diccionario.

Siguiendo con este enfoque aparece la posibilidad de no tener ya que preocuparnos por escoger celdas particulares de memoria, sino dejar esta responsabilidad al propio traductor. Esto es, eximir al programador de la tarea de escoger celdas de memoria y asignársela al traductor T1. Es en este momento cuando hay que introducir el concepto de variable. Una variable será un nombre simbólico asociado con una celda cualquiera de la memoria de la computadora, sólo que esta asociación se hará de manera automática.

Así, en lugar de decidir si se cargará el acumulador con la casilla 21 (o, si fuera el caso, con la 3456 o la 19689, etc.), se escribirá

```
CARGA_Ac ALFA
```

donde ALFA es el nombre simbólico de una celda de memoria (y ya no importa cuál será ésta). Mientras el traductor T1 reconozca siempre que ALFA corresponde a una celda en particular (escogida por él mismo) no habrá problemas. Nosotros, como programadores, diremos ALFA y el traductor le dirá a la computadora 21 (o, si fuera el caso, 3456 ó 19689, etc.).

Independencia de las direcciones de la memoria: manejo simbólico

Este es un paso de fundamental importancia; lograrlo implica estar, efectivamente, "por encima" de la memoria, al no tener que preocuparse por direcciones absolutas. Es decir, como programadores, trabajaremos en un

ambiente simbólico, no absoluto. ¿Cuáles son las ventajas de esto? En primer lugar, empezar a desligarse de la computadora, y dirigirse a ella en un lenguaje más simbólico que antes. Además, los programas son mucho más flexibles pues, por ejemplo, si la celda 21 está ocupada por otro usuario, el traductor T1 podrá asignar ALFA a cualquier otra celda que esté libre. Todo esto permite la creación de programas más legibles para un ser humano y, por tanto, más útiles.

El programa, entonces, dirá:

```
CARGA_Ac        ALFA
SUMA            BETA
GUARDA_Ac       GAMA
ALTO
```

que ya es algo más parecido a lo que realmente se desea hacer, que es:

```
GAMA = ALFA + BETA
```

en donde, por ejemplo, ALFA vale 5 y BETA vale 7.

Veamos qué cambios habrá que hacer a T1 para que pueda, por sí solo, escoger y asignar celdas de memoria a las variables simbólicas. Claro que tendrá que guardar las direcciones absolutas que asignó a las variables simbólicas, para poder reconocerlas en el momento que se requieran. Por ejemplo, si se intentara elaborar un programa para calcular $C = A + B$ y luego $D = E - C$, se escribiría algo como:

```
GARCA_Ac        A
SUMA            B
GUARDA_Ac       C
CARGA_Ac        E
RESTA           C
GUARDA_Ac       D
ALTO
```

Resulta claro que el traductor debe reconocer las direcciones de todas las variables simbólicas ya que, por ejemplo, C es requerida dos veces a lo largo del programa.

Asimismo, habrá que ocuparse del manejo de etiquetas, que son referencias simbólicas a renglones dentro del programa y se usan para realizar "brincos condicionales" necesarios, por ejemplo, para modificar el flujo de la ejecución del programa dependiendo de los valores que tomen ciertas variables. Como se vio, la máquina puede comparar el contenido de dos registros y determinar cuál es mayor con respecto al otro. Con esta información, se decide qué parte del programa debe ejecutarse a continuación, "brincando", de esta manera, partes del código. En el siguiente capítulo se estudian más detalladamente las estructuras de control dentro de un programa. Por lo pronto, basta con saber que el funcionamiento de los lenguajes de alto ni-

vel depende en buena parte de estas etiquetas, que el compilador usará internamente.

La existencia de las etiquetas obliga a que el proceso de traducción se cumpla en dos pasos sobre el texto del programa fuente: 1) guardar las definiciones de etiquetas en una tabla especial y 2) reemplazar las referencias a ellas por las direcciones donde se encontraron.

El nuevo acercamiento será:

```
! Programa T1, segunda versión
! -primer paso-
Para cada renglón del programa fuente ejecutar lo siguiente:
  Si existe una etiqueta, guardar su dirección en la tabla.
! -segundo paso-
Regresar al primer renglón.
Para cada renglón del programa fuente ejecutar lo siguiente:
  Buscar la palabra mnemónica en el diccionario.
  Si está, entonces traducirla a código objeto.
     En caso contrario, marcar error.
  Localizar la variable simbólica de ese renglón.
  Si existe, entonces preguntar si es la primera vez que se encuentra.
  Si así es, asignarle una dirección absoluta y guardarla en un nuevo diccionario
  (tabla de símbolos); escribir esa dirección en el programa objeto.
     En caso contrario, determinar si se trata de una variable simbólica
     predefinida (que deberá ser reemplazada por su dirección absoluta),
     o de una referencia a una etiqueta (que será reemplazada por la
     dirección que está guardada en la tabla).
```

Si se aplica este procedimiento para el programa recién descrito —aunque no tiene etiquetas—, el traductor T1 producirá los siguientes resultados:

	Programa fuente		Programa objeto obtenido	
1	CARGA_Ac	*A*	21	70
2	SUMA	*B*	57	71
3	GUARDA_Ac	*C*	96	72
4	CARGA_Ac	*E*	21	73
5	RESTA	*C*	42	72
6	GUARDA_Ac	*D*	96	74
7	ALTO		70	

Se supone que T1 decidió asignar celdas de memoria a partir de la dirección 70 (aunque pudo haber puesto cualquier otro número). La tabla de símbolos para este programa (producida internamente por T1) fue:

Variable simbólica	Dirección asignada
A	70
B	71
C	72
E	73
D	74

Obsérvese que, dado que la variable *C* aparece dos veces en el programa fuente, su dirección absoluta correspondiente (72) también aparece dos veces. No hay que confundir, sin embargo, las dos apariciones del número 70. La primera vez que aparece es porque representa la dirección que el traductor asignó a la variable *A*, mientras que la segunda representa la codificación (de acuerdo con el diccionario usado desde la página 34 de la instrucción ALTO.

Cuando el traductor T1 está leyendo el renglón fuente número 3 se encuentra por primera vez con la variable *C*, por lo que la introduce en la tabla de símbolos y la reemplaza, en el código objeto, por la dirección absoluta 72 recién asignada. Cuando la encuentra de nuevo, en el renglón 5, no la introduce en la tabla, pues ya está ahí; ahora, simplemente la reemplaza en el código objeto por su dirección, 72.

En este momento ya es posible llamar al traductor T1 por su verdadero nombre: ensamblador. Un **ensamblador** es, entonces, un traductor que asigna direcciones absolutas a las variables simbólicas que el programador escogió, liberándolo de esa tarea.

Los programas producidos de esta manera se conocen como programas escritos en **lenguaje ensamblador**. Es evidente que es mucho más conveniente escribir programas en ensamblador que en lenguaje de máquina. Para información sobre los programas ensambladores pueden consultarse las referencias [BECL85], [DONJ72] y [ULLJ76].

4.3 Macroprocesadores

Se podría pensar también en dar al ensamblador la capacidad de repetir, por medio de una orden, grupos completos de instrucciones que deben aparecer en múltiples ocasiones. Esto es, compactar renglones repetitivos en uno solo que fungirá como su abreviatura, y pedir al ensamblador que lo expanda a la hora de la traducción.

Si los renglones

```
CARGA_Ac    A
SUMA        B
GUARDA_Ac   C
```

aparecen con frecuencia en un programa en ensamblador, se podrían agrupar en uno solo que se llamara, por ejemplo, **ADICIÓN**. Cada vez que el ensamblador observara el mnemónico **ADICIÓN** lo expandiría para producir los tres renglones anteriores.

Este nuevo esquema recibe el nombre de **macroprocesamiento**, y es de importancia capital dentro de las ciencias de la computación porque permite —en su expresión más general— la sustitución textual de símbolos de un tipo con símbolos de otro. Esto, como se verá en la sección 5.3, forma parte de la idea central de la computación.

Un macroprocesador* trabaja con definiciones de renglones (o de símbolos) llamadas **macros** (o macrodefiniciones), que serán expandidas cuando se las llame. Una macrollamada, por tanto, será la invocación de una macrodefinición (por su nombre) para producir nuevos renglones de texto.

Para convertir los tres renglones anteriores en una macrodefinición, habrá que encerrarlos entre las palabras **MACRO**, seguida del nombre asignado, y **FIN_MACRO**, de la siguiente manera:

```
  MACRO        ADICIÓN
CARGA_Ac         A
SUMA             B
GUARDA_Ac        C
FIN_MACRO
```

Cuando se requiera llamarla, sólo habrá que nombrarla.

Así, como se dijo antes, cuando el ensamblador detecte el nombre **ADICIÓN** producirá los tres renglones que hacen la operación simple $C = A + B$.

Para hacer la operación $K = L + W$, por ejemplo, existen dos posibilidades: escribir los mismos tres renglones, cambiando sólo las letras (variables), o bien usar la misma macrodefinición, pero permitiendo la existencia de ciertos elementos variables, llamados **parámetros**.

Macrodefiniciones con parámetros

La macro dirá ahora

```
MACRO ADICIÓN (3)
CARGA_Ac    ?1
SUMA        ?2
GUARDA_Ac   ?3
FIN_MACRO
```

Si se invoca como **ADICIÓN** (*A*, *B*, *C*) se obtendrá exactamente el mismo resultado que antes, pero si se llama como **ADICIÓN** (*L*, *W*, *K*), se obtendrá la nueva operación.

---

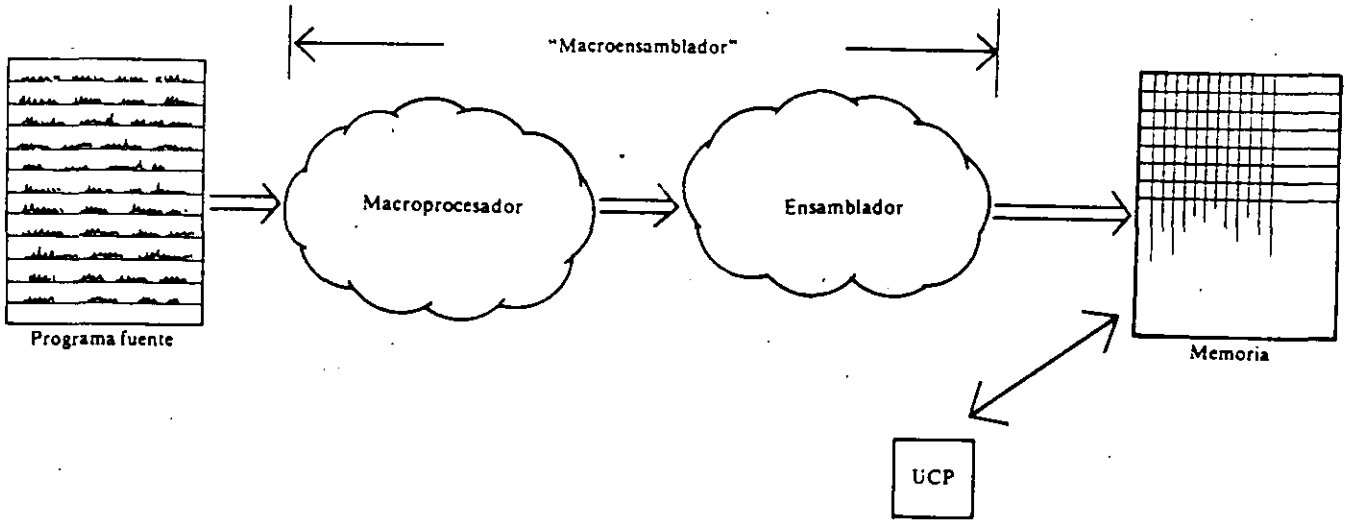
* Este término hace referencia a un programa (o conjunto de programas), y no debe confundirse con un dispositivo físico, como el microprocesador.

Para que un macroprocesador pueda manejar parámetros, requiere una tabla en donde guardarlos, para luego hacer la correspondencia entre los parámetros ficticios (representados por ?1, ?2 y ?3 del ejemplo) y los parámetros reales (o **argumentos**) *A*, *B*, *C* de la primera llamada, o *L*, *M*, *K* de la segunda.

A los ensambladores que tienen integrado un macroprocesador se les conoce como **macroensambladores**.

No hay lugar aquí para seguir con este tema, por lo que se remite al lector interesado a las referencias [BROP75] y [KERB76].

Sólo falta por averiguar, con cierto nivel de detalle, qué se requiere para que la computadora pueda traducir los programas fuente a programas objeto por sí sola.



Función de un macroensamblador

4.4 Cargadores

Para que la computadora haga la traducción de los programas escritos en lenguaje ensamblador a lenguaje de máquina, será necesario que el programa traductor resida, ya traducido, en la memoria. Esto es, que la computadora ejecute el programa T1 para que traduzca los programas fuente a programas objeto.

Supóngase que alguien escribió un programa para resolver un conjunto de ecuaciones; ese programa está escrito en lenguaje ensamblador, y lo llamaremos P1. Se estudia ahora la serie completa de pasos que hay que realizar para lograr que la máquina traduzca y ejecute P1. (Se inicia desde que la computadora está apagada y se analiza lo que hay que hacer para llegar al final.)

El problema de la carga

Cuando se enciende por vez primera la computadora, la memoria está completamente vacía; es decir, el procesador está detenido, esperando alguna instrucción en memoria para leer, decodificar y ejecutar. ¿Cómo se saca a la computadora de este letargo? Está claro que aún no es posible ejecutar algún programa, por la sencilla razón de que no existe ninguno residente en la memoria. Lo que hay que hacer es, pues, cargar algún programa en memoria para poder ejecutarlo. Y aquí encontramos el primer problema: ¿cómo se mete un programa a la memoria? Existen dos posibles respuestas; la primera consiste en cargar manualmente celdas de memoria con valores numéricos (que representen la codificación, en lenguaje de máquina, de algún programa), y la segunda en ejecutar un programa que sirva para hacer esto de manera automática.

Exploremos la primera posibilidad. Supóngase que el programa fuente P1 tiene cien renglones de longitud, y que cada renglón tiene la misma forma general que el programa de la página 82. Esto significa un gran esfuerzo, ya que habría que meter manualmente varios miles de ceros y unos a varios cientos de celdas de la memoria. Si ya se hubiera realizado este penoso paso todavía quedaría la enorme molestia de tener que cargar también manualmente el traductor T1 completo.

La segunda posibilidad es mucho más prometedora, y consiste en escribir un programa para que haga estos pasos por nosotros. Se llamará **cargador**. Las funciones de un cargador son relativamente sencillas, y consisten en extraer información objeto de algún medio externo a la memoria (tarjetas perforadas, disco o cinta magnética, por ejemplo) y colocarla en celdas sucesivas de la memoria, a partir de una celda preespecificada.

He aquí el diseño básico de un cargador:

```
! Programa cargador, primer acercamiento.
Localizar el dispositivo de memoria secundaria que contenga la información
que se va a cargar.
Averiguar cuál es la casilla de memoria a partir de la cual va a quedar cargado
ese programa objeto.
Para cada renglón del programa objeto ejecutar lo siguiente:
  Determinar cuántas celdas de memoria se requieren para almacenar
  esos datos binarios.
  Depositar esos datos en celdas contiguas de la memoria, a partir de la
  celda inicial.
  Tomar nota de la última celda utilizada, y considerarla como si fuera la inicial.
```

Si se logra traducir este programa a lenguaje de máquina y se deja en la memoria, tendremos ya una herramienta muy poderosa, con la que se puede cargar en memoria cualquier programa objeto, siempre y cuando se le indique dónde está almacenado (en disco o cinta) el programa objeto, y a partir de cuál celda de memoria se desea que lo deposite.

El dilema de la carga inicial de un programa

Esto, sin embargo, da lugar a otro problema: ¿cómo se carga el cargador?

La respuesta ya no puede ser "por medio del cargador", porque está claro que éste no puede cargarse a sí mismo; para ello tendría que estar residente

en memoria (para que la UCP lo pudiera ejecutar), ¡y éste es precisamente el problema que se quiere resolver!

Nos encontramos ante un problema que no tiene solución, por lo menos, en términos de un programa que lo resuelva. Este problema inicial —romper el círculo vicioso recién descrito— recibe el nombre de *bootstrap*, que en inglés significa algo así como "el problema de tratar de levantarse del suelo tirando de las cintas de nuestras propias botas". Éste es, evidentemente, un problema que tiene "truco": requiere medios externos para poderse resolver.

El truco consiste en cargar a mano el cargador para evitar los obstáculos lógicos mencionados. Sólo que esta operación se tendría que repetir cada vez que se encienda la computadora, ya que cuando se retira la corriente eléctrica la memoria pierde todos sus contenidos. La otra solución a este problema resulta muy interesante y es, a grandes rasgos, la siguiente.

Se escribe otro pequeño programa (que llamaremos "minicargador"), cuya única función consiste en cargar el cargador. Este miniprograma no será de uso general, y solamente servirá para extraer al cargador objeto de un lugar preestablecido (de una sección de un disco magnético determinado, por ejemplo) y depositarlo en una zona también preestablecida de la memoria central. Luego de hacer esto, el minicargador se desactiva y cede el control al cargador. Como este programa es de uso particular y cumple una sola función muy específica, será pequeño (unas pocas decenas de renglones fuente), por lo que será posible traducirlo a mano al lenguaje de máquina.

La situación aparece ahora así: cuando se enciende la computadora, se carga manualmente el minicargador objeto y se ejecuta. Éste, a su vez, cargará al cargador, y a partir de ahí se podrá seguir con el proceso.

Es obvio que, en una computadora normal, el minicargador no se carga manualmente, sino por medios electrónicos, utilizando una memoria especial tipo ROM, la cual deposita automáticamente su contenido en la memoria central. Al proceso de cargar el minicargador y, con ello, dar "vida" a la computadora se le conoce como IPL (*initial program load*, carga del programa inicial). El IPL se ejecuta cada vez que se enciende la computadora y es el requisito previo para la operación de la misma.

Los pasos para traducir y ejecutar el programa P1 son, entonces:

1. Dar IPL, para cargar el cargador
   (no es necesario si la computadora ya está operable).
2. Cargar el traductor T1 objeto, que deberá estar residente en algún lugar de la memoria secundaria.
3. Ejecutar T1 para que lea el programa fuente P1 (que estará residente también en algún disco o cinta magnética) y lo traduzca a lenguaje de máquina, dejándolo en una sección determinada de la memoria real.
4. Ejecutar el programa objeto P1 resultante.

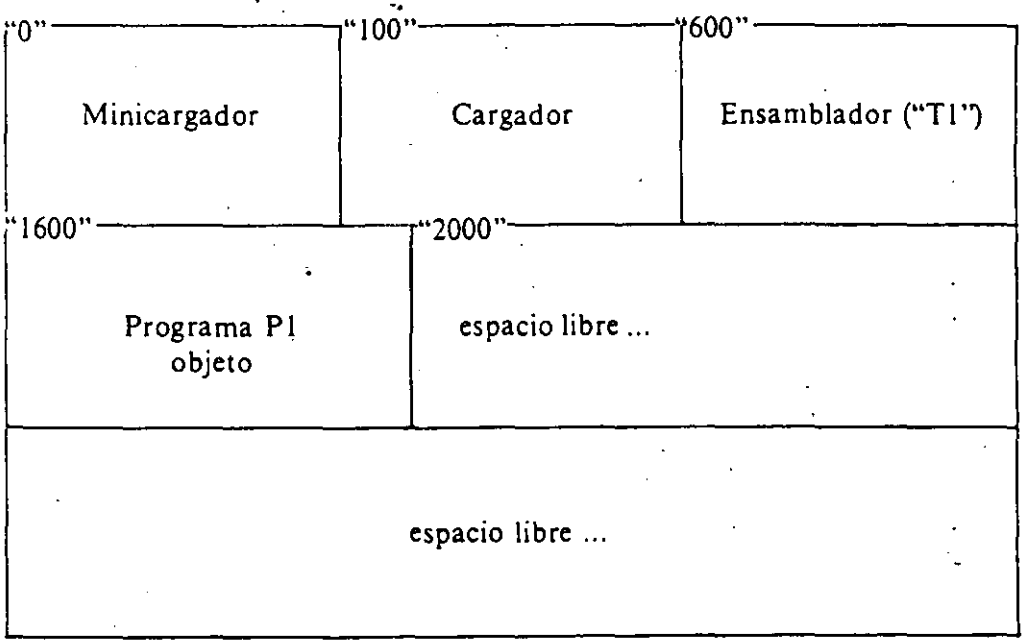
Veamos estos pasos con más detalle. Supóngase que el programa ensamblador objeto T1 reside en un cierto disco magnético DM-T1, y que el programa fuente P1 está en otro disco magnético, DM-P1.

Supóngase también que el minicargador residirá en memoria a partir de la celda cero, y que mide 100 bytes. El cargador, ya traducido, medirá 500 bytes.

El ensamblador objeto T1 mide 1000 bytes, y el programa objeto P1 medirá, una vez ensamblado, 400 bytes.

Si se efectúan todas las operaciones de carga de manera secuencial y utilizando celdas contiguas de memoria, ésta se verá así (justo antes de ejecutar el paso 4 anterior):

MEMORIA CENTRAL



Este diagrama (que no está a escala) indica cuáles programas objeto están residentes en la memoria, y a partir de cuál celda.

Teniendo en cuenta las direcciones descritas para asignarlas al contador de programa (CP), los pasos anteriores serán:

0. Dar IPL (si es necesario).

Pasos para la ejecución de un programa en una computadora

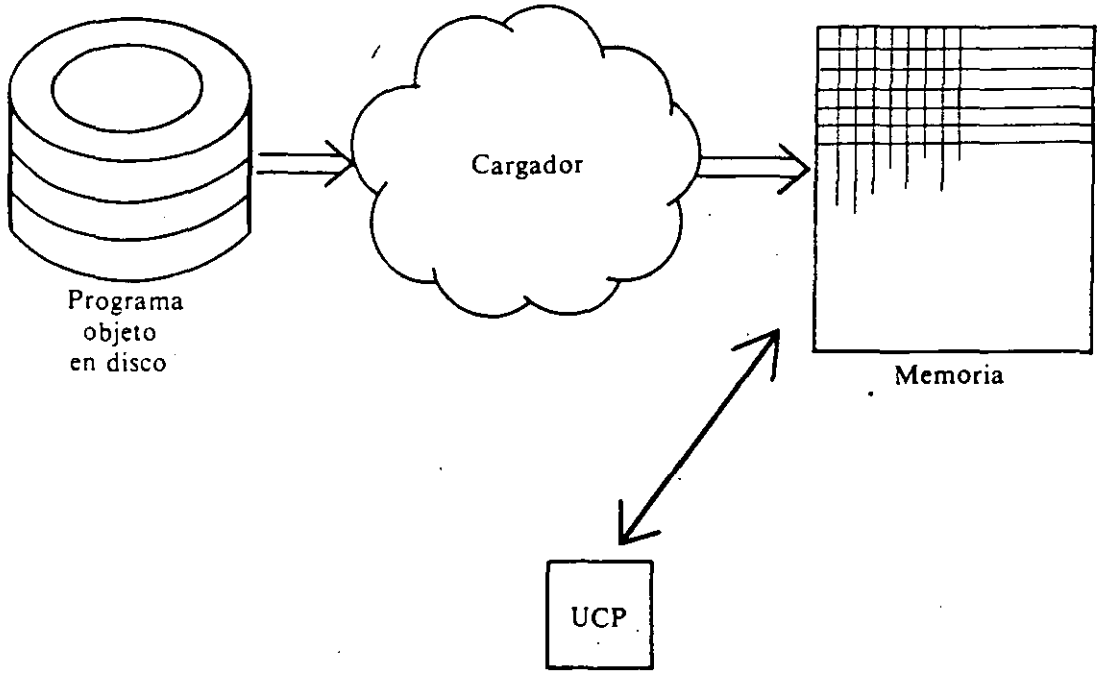
1. CP ← 100
   (Localizar el disco DM-T1 y ejecutar el cargador, para que deposite su contenido en memoria, a partir de la celda 600.)

2. CP ← 600
   (Localizar el disco DM-P1 y ejecutar el ensamblador, para que deposite el resultado de la traducción a partir de la celda 1600.)

3. CP ← 1600
   (Esto es, ejecutar el programa objeto P1.)

Será responsabilidad del programa P1 ejecutar la instrucción ALTO como último paso; de no hacerlo así, la computadora tratará luego de ejecutar lo que contenga la celda 2000, y eso tendría resultados impredecibles.

Más adelante se verá cómo se pueden convertir los pasos 1 a 3 en un programa para que, una vez traducido a lenguaje de máquina, gobierne la operación de la computadora y nos libere de esa tarea. Un programa de este tipo recibirá el nombre de monitor



Función de un cargador

4.5 Compiladores

Los capítulos anteriores han dado la posibilidad de escribir programas fuente para aplicaciones particulares en lenguaje ensamblador, ya que se dispone de un traductor T1 que los traducirá para la máquina a su propio lenguaje binario. Se puede también dejar estos programas fuente residentes en dispositivos de memoria auxiliar, y cargarlos a voluntad por medio del cargador, que también hemos diseñado. Estamos, pues, en una situación bastante buena, donde se tiene un adecuado nivel de comunicación con la computadora. No obstante, se desea aun más capacidad, y una mayor flexibilidad para comunicar los requerimientos a la máquina. Lo que se quiere lograr ahora es la posibilidad de comunicarse con la computadora en un lenguaje incluso más parecido al nuestro (y que, por tanto, será menos parecido al lenguaje de unos y ceros).

¿Qué se puede hacer?

Exploremos las posibilidades de intentar comunicarse con la computadora usando un lenguaje de más alto nivel expresivo. Cuando se dice "lenguaje de alto nivel" se piensa en uno que permita, con una sola orden, decir cosas complejas; esto es, un lenguaje dotado de una estructura, que le dé soporte a lo que recién se dijo.

Necesariamente hay que pensar entonces en el problema de la traducción de lenguajes de alto nivel expresivo, porque lo que se desea hacer es comunicarse con la máquina en un lenguaje de este tipo y esperar que reciba nuestros mensajes mediante un traductor que lo convierta a su lenguaje de máquina.

El problema de la traducción

Analicemos el caso de traducir del español al inglés, por ejemplo, la siguente frase: la casa es azul.

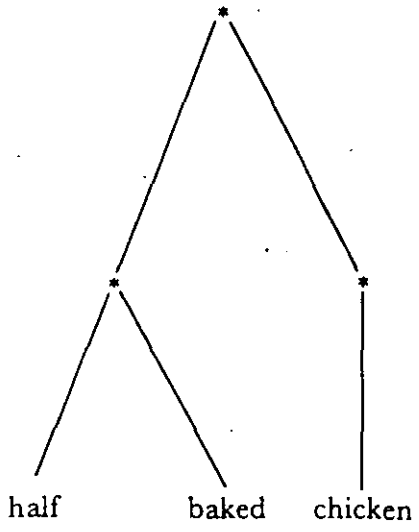
Si se usa un diccionario, se encontrará que "la" se traduce por *the*, "casa" por *house*, "es" por *is* y "azul" por *blue*, por lo que la frase ya traducida será *the house is blue*.

Está claro, sin embargo, que esto no fue más que una simple casualidad. Inténtese un ejemplo más complejo y se encontrarán de inmediato las dificultades inherentes a todo proceso de traducción de lenguajes de alto nivel expresivo; es decir, un simple diccionario no basta para lograr una buena traducción y, a veces, ni siquiera para lograr algo que medianamente se asemeje a la frase original. Esto se debe, por supuesto, a la **estructura del lenguaje**, que está definida por su gramática, y a un mundo de significados que no es reducible a un diccionario.

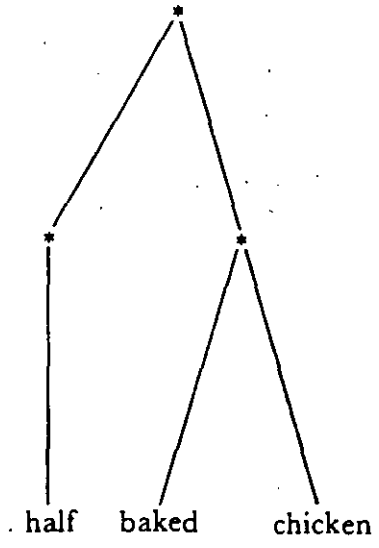
Así, aunque se haya traducido la frase "sufragio efectivo, no reelección" por medio de un diccionario, nos veremos en dificultades casi insalvables cuando se intente traducir la frase "sufragio efectivo no, reelección", que aunque tiene exactamente los mismos componentes (las mismas cuatro palabras y una coma) significa nada menos que lo contrario que la frase original. Ningún diccionario será suficiente para dar cuenta de la diferencia, ya que éstos trabajan únicamente con palabras aisladas, sin tomar en cuenta la estructura gramatical.

Noam Chomsky, lingüista del Instituto Tecnológico de Massachusetts (MIT), que en 1956 publicó un estudio ya clásico sobre gramáticas formales (esto es, estudiadas desde un punto de vista matemático), proponía otro ejemplo, esta vez en inglés, para poner en evidencia que en toda frase de un lenguaje existe como respaldo una estructura que le da forma y sentido. ¿Qué significa la frase *half baked chicken*? Puede significar tanto "medio pollo cocido" como "pollo medio cocido", que de ninguna manera quiere decir lo mismo. Esta es una clásica frase ambigua. La *ambigüedad* está determinada por la estructura que respalda la frase, como a continuación se explica.

Para el caso de "pollo medio cocido", la estructura de la frase es como sigue:



Mientras que para "medio pollo cocido" es:



Diagramas de estructura gramatical

Sin preocuparse todavía por el significado de los diagramas, sí diremos que existen varias maneras de agrupar las palabras (por medio de lo que en lingüística se conoce como "estructura profunda"), y que estas maneras le confieren significados distintos a la misma frase.

A continuación se verá cuál es la secuencia para lograr la traducción de frases dotadas de estructura interna.

El paso inicial consiste en **reconocer** todos y cada uno de los **símbolos** aislados que constituyen la frase; lo que, a su vez, implica reconocer las letras (y signos de puntuación) y reconocer las palabras. Obsérvese que reconocer no necesariamente significa entender; para reconocer un símbolo lo único que se requiere es buscarlo (y encontrarlo) en un diccionario previamente especificado.

El proceso de la traducción

Se llama a esta primera etapa **análisis lexicográfico**

Una vez concluido este análisis se llega a la parte interesante, *encontrar la estructura gramatical* de la frase cuyos elementos ya se reconocieron. Este proceso es complejo, y requiere de múltiples análisis que utilizan métodos matemáticos para lograr develar la estructura inherente a la frase. La idea general consiste en tratar de acomodar alguna estructura gramatical apropiada para la frase objeto del análisis, guiándose por medio de las palabras que la componen. Más adelante se da un ejemplo.

A esta segunda etapa se le llama **análisis sintáctico**

Terminada la fase sintáctica o gramatical se está ya en posición de entender lo que la frase significa, por medio del **análisis semántico**

El **compilador** (que es el nombre de este nuevo traductor) tendrá entonces que hacer estos tres tipos de análisis sobre las cadenas de entrada (esto es, sobre el programa fuente), para poder llegar a traducirlo al lenguaje de máquina o, por lo menos, al lenguaje ensamblador y obtener finalmente el mismo programa pero ya en lenguaje objeto.

La teoría matemática requerida para el diseño de un compilador rebasa los límites de este curso (como se describe, por ejemplo, en [AHOA77], [AHOA85], [HOPJ79] y [TREJ85]) por lo que tan sólo se estudiará el proceso de compilación desde un punto de vista muy general.

La tarea central del analizador lexicográfico consiste en separar los **componentes léxicos** (o *tokens*) de entre el conjunto de símbolos del programa fuente. Esto es, en un renglón común coexisten símbolos de diversas clases (letras, dígitos, símbolos de puntuación, blancos y caracteres especiales) aunque sean invisibles, y es necesario aislar los componentes sintácticos de este conglomerado de caracteres. Para nosotros es obvio que la frase "uno, dos, tres" consta de tres palabras, pero en realidad contiene catorce símbolos diferentes que es necesario agrupar de alguna manera. Esta es, a grandes rasgos, la razón de ser del análisis léxico.

El modelo matemático de un analizador de este tipo recibe el nombre de autómata finito (descritos, por ejemplo, en [HOPJ69]). Un autómata de este tipo es una función matemática que puede reconocer grupos de caracteres que constituyen un componente sintáctico. En el siguiente capítulo se hace una breve referencia a este tipo de construcciones matemáticas.

Sin embargo, el problema del análisis léxico es sencillo comparado con el que le sigue, el análisis sintáctico, que no fue entendido sino hasta hace algunos años; todavía en la actualidad existen aspectos oscuros sobre su funcionamiento general. Los analizadores sintácticos (*parsers*) se dividen en dos grandes familias: los que funcionan en forma "ascendente" y sus contrarios, en forma "descendente". Para poder discutir estos puntos, aunque sea mínimamente, será necesario mencionar antes algunos elementos sobre la teoría de las gramáticas y los lenguajes formales*. Como se dijo, esta teoría nace en la década de 1950, y trata sobre las propiedades de ciertas construcciones formales llamadas **gramáticas**, que no son sino formulaciones matemáticas de la estructura de los lenguajes formales, de la misma forma que la gramáti-

* En el capítulo 5 se dedica una sección a este tema.

ca que todos aprendimos en la escuela elemental describe la estructura del lenguaje ordinario.

El problema de la comunicación

Para caracterizar el problema de la comunicación se puede pensar que una gramática es un *generador* de palabras (o frases), que luego llegarán a un *reconocedor*, que se encargará de *decidir* si una frase es "hija legítima" de cierta gramática o no; esto es, el reconocedor deberá hacer cierto análisis sobre las frases recibidas y proceder luego a interpretarlas.

A continuación hay un ejemplo muy elemental, sobre un subconjunto de la gramática del español.

Todo hablante de nuestro idioma reconocerá que la frase "la casa es azul" es correcta desde cualquier punto de vista (lexicográfico, sintáctico y semántico). Para analizar el porqué de ello se partirá del conocimiento de que una oración (o frase) está compuesta de palabras (que a su vez constan de letras). Las palabras, sin embargo, se clasifican en ciertos tipos gramaticales que luego formarán construcciones más complejas. En suma:

- *la* es un ARTÍCULO.
- *casa* es un SUSTANTIVO.
- *es* es un VERBO.
- *azul* es un ADJETIVO.

y

- Un ARTÍCULO seguido de un SUSTANTIVO es una FRASE NOMINAL.
- Un VERBO seguido de un ADJETIVO es una FRASE VERBAL.
- Una FRASE NOMINAL seguida de una FRASE VERBAL es una ORACIÓN.

O más escuetamente,

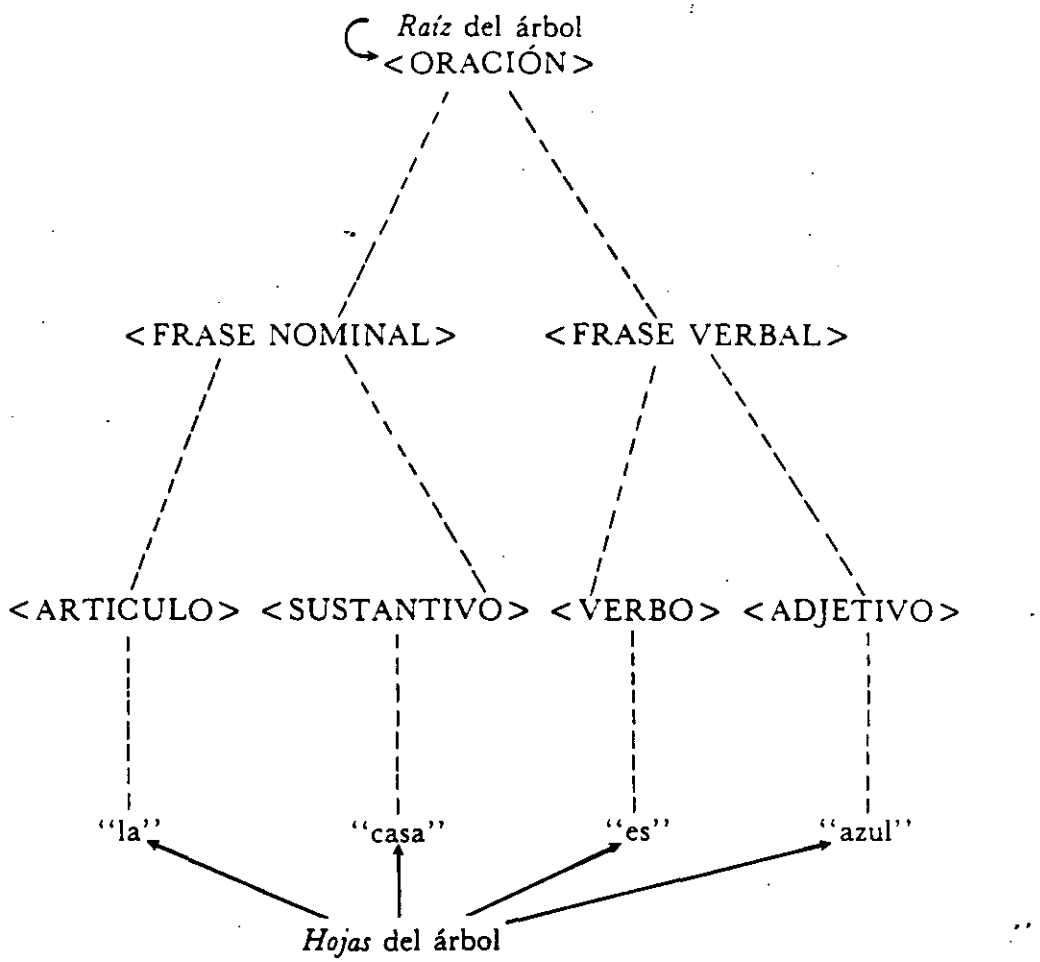
1) <ORACIÓN> → <FRASE NOMINAL> <FRASE VERBAL>
2) <FRASE NOMINAL> → <ARTÍCULO> <SUSTANTIVO>
3) <FRASE VERBAL> → <VERBO> <ADJETIVO>
4) <ARTÍCULO> → "la"
5) <SUSTANTIVO> → "casa"
6) <VERBO> → "es"
7) <ADJETIVO> → "azul"

Estas siete reglas gramaticales configuran una primera gramática formal, con la que se trabajará un poco. Obsérvese que se están usando algunos símbolos nuevos: con las llaves triangulares se encierran palabras de la gramática que se llamarán **no terminales**, mientras que con las comillas se distinguen las palabras **terminales**. Estas últimas son las que forman las frases terminales, o sea, los elementos finales de una construcción gramatical, que son los únicos que se muestran al mundo exterior. Una frase u oración tiene una estructura interna, y los elementos que se usan para definir esta estructura profunda de la frase son precisamente los no terminales.

El otro elemento nuevo es la flecha, que liga miembros izquierdos (no terminales), con miembros derechos (que pueden ser terminales o no).

La regla 6) <VERBO> → "es" se lee: "el no terminal VERBO *produce* el terminal 'es' ".

El lector podrá comprobar que la frase "la casa es azul" tiene la siguiente estructura:



Ahora se pueden entender un poco mejor los conceptos del análisis sintáctico. Si partimos de la parte superior de este diagrama (que recibe el nombre de árbol sintáctico) y se intenta aplicar una a una las reglas de producción —son siete—, se obtiene lo siguiente:

<ORACIÓN> ⇒ <FRASE NOMINAL> <FRASE VERBAL>
⇒ <ARTÍCULO> <SUSTANTIVO> <FRASE VERBAL>
⇒ "la" <SUSTANTIVO> <FRASE VERBAL>
⇒ "la" "casa" <FRASE VERBAL>
⇒ "la" "casa" <VERBO> <ADJETIVO>
⇒ "la" "casa" "es" <ADJETIVO>
⇒ "la" "casa" "es" "azul"

Un análisis sintáctico

(La doble flecha se lee "genera mediante la aplicación de una regla de producción", o simplemente "genera").

Obsérvese que se partió del tope (o raíz) del árbol y se llegó a las hojas terminales. Éste fue el primer análisis sintáctico descendente; aunque trivial, es representativo.

¿Cómo será el análisis sintáctico ascendente? Hagamos lo siguiente:

"la" ⇐ <ARTÍCULO>
"casa" ⇐ <SUSTANTIVO>
<ARTÍCULO> <SUSTANTIVO> ⇐ <FRASE NOMINAL>
"es" ⇐ <VERBO>
"azul" ⇐ <ADJETIVO>
<VERBO> <ADJETIVO> ⇐ <FRASE VERBAL>
<FRASE NOMINAL> <FRASE VERBAL> ⇐ <ORACIÓN>

Obsérvese que ahora se procedió exactamente a la inversa: partiendo de los elementos terminales se encontró un camino que lleva hasta la raíz del árbol.

Aunque en apariencia el análisis ascendente no es más que el inverso del descendente, en realidad es mucho más complejo. La razón de lo anterior reside, intuitivamente, en que el problema central del análisis descendente consiste en escoger alguna regla y aplicarla, partiendo de su miembro izquierdo (que consta de un solo elemento no terminal), mientras que en el caso contrario hay que escoger alguna regla y "desaplicarla"; pero ahora ya no existe un solo elemento del lado derecho, sino varios y, por tanto, aumenta la gama de combinaciones posibles.

De hecho, hace apenas quince años que se encontraron algoritmos eficientes para realizar análisis sintácticos ascendentes, mientras que los descendentes fueron inventados hace más de treinta*. ([GRID71], que es un buen libro de compiladores, por ejemplo, no recogió estos últimos.)

Con este somero análisis sobre el análisis sintáctico pasamos a la siguiente etapa de un compilador: el análisis semántico. Aquí lo importante es determinar la coherencia entre lo que se dice, por medio de un lenguaje, y los elementos del mundo a los cuales se está haciendo referencia. En este caso, "el mundo" es, por supuesto, la computadora, sus registros, sus celdas de memoria, etc. El análisis semántico averigua, por ejemplo, si una expresión dentro de un programa de computadora significa algo, y no pide hacer una operación aritmética sobre una cadena de caracteres, cosa que no tendría sentido.

En muchas referencias sobre compiladores se llama fase semántica a la parte que se encarga de lo recién descrito, pero además suele incluirse también allí las funciones de generación de código que ahora se explican.

---

* Las ideas originales sobre análisis sintáctico ascendente se deben a Donald Knuth, en el artículo "On the Translation of Languages from Left to Right", aparecido en la revista *Information and Control*, octubre, 1965; pero no fue sino hasta la publicación del artículo "An Efficient Context-Free Parsing Algorithm", que es un resumen de la tesis doctoral de Jay Earley, publicado en *Communications of the Association for Computing Machinery*, febrero, 1970, cuando se contó con un algoritmo práctico, que luego se ha refinado aun más.

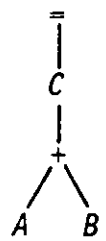
Ya que se ha analizado a fondo una frase y se ha determinado su validez lexicográfica, sintáctica y semántica, hay que traducirla. Es importante observar que la operación de traducción es lógicamente posterior a las operaciones de análisis.

La traducción —o **generación de código**— busca representar la frase fuente original en términos de elementos de un lenguaje mucho más sencillo, que ya no está dotado de estructura. O sea, precisamente, traducir la frase fuente al lenguaje de máquina (o por lo menos al lenguaje ensamblador).

En este nuevo ejemplo se muestra eso. Supóngase que se desea efectuar la operación `C = A + B` donde las letras representan variables de tipo numérico. Un renglón de este programa fuente será, entonces,

```
C = A + B
```

El compilador efectuará el análisis lexicográfico y luego el sintáctico, para determinar que la estructura gramatical de la frase es



Hecho esto, procederá a determinar si las tres letras son variables numéricas definidas dentro del programa fuente. El siguiente paso es aplicar ciertas reglas predefinidas para la generación de código. Estas reglas indicarán las operaciones por efectuar para traducir las expresiones primitivas que el análisis sintáctico determinó como componentes de la frase fuente original. Para el ejemplo, la estructura del árbol sintáctico pide que se efectúen —en ese orden— las siguientes reglas de generación de código intermedio:

```
1. +   (X1, A, B)
2. =   (C, X1, -)
```

La estructura de cada regla es como sigue: se especifica la operación que se aplicará al triplete de elementos dentro del paréntesis. El primer elemento recibirá el resultado de la operación que se efectuará sobre los dos siguientes. La regla número uno especifica que se haga la suma de `A` y `B` y se deje el resultado en un elemento temporal `X1`. La segunda regla hace la transferencia de este elemento temporal a la variable `C`, y deja indefinido el tercer elemento, puesto que no se requiere.

Aunque en el ejemplo está claro que este elemento temporal sobra, obsérvese que esto no puede estar previsto en las reglas predefinidas, que por ser de carácter general, no pueden saber que se desea dejar el resultado de la suma en la variable `C`.

A estas alturas es evidente que los elementos a los que nos estamos refiriendo no pueden ser otra cosa más que celdas de la memoria de la computadora. Toca ahora seleccionar las celdas que se van a utilizar, y se usarán las que estén disponibles. Por lo pronto no importa mucho cuáles serán, sino cuántas y en qué orden.

A la generación de código recién escrita se le conoce como **generación de código intermedio**, ya que no se trata de lenguaje de máquina, sino de una especificación relativamente informal, en términos de elementos (temporales o no) que son producidos "a ciegas" por medio de reglas predefinidas.

Una operación importante que se puede hacer sobre el código intermedio recién generado es la **optimización**. Una vez generadas las líneas de código intermedio es posible "observarlas desde lejos" y tratar de eliminar redundancias y repeticiones. Nótese que esto no se puede hacer al tiempo de generación de código intermedio, porque equivaldría a quitar a las reglas predefinidas su carácter genérico.

En el siguiente ejemplo se ve la conveniencia de mantener la generalidad de las reglas de traducción, que manejan elementos temporales. Del lado izquierdo aparecen las frases fuente, y del derecho sus correspondientes traducciones a código intermedio.

```
C = A + B          1. + (X1, A, B)
                   2. = (C, X1, –)
C = 0              3. = (C, 0, –)
D = E – (A + B)    4. – (X2, E, X1)
                   5. = (D, X2, –)
```

Independientemente del objetivo que tenga ese programa, conviene manejar los elementos temporales libremente, y sólo decidir si se eliminan o no en una segunda fase.

En todo caso, luego se abordará el problema de convertir el código intermedio (quizá ya optimizado) en código objeto propiamente dicho; esto es, en lenguaje de máquina. A esta nueva fase se le conoce como **generación de código objeto**.

Se describirá ahora la generación de código objeto a partir del código intermedio del primer ejemplo:

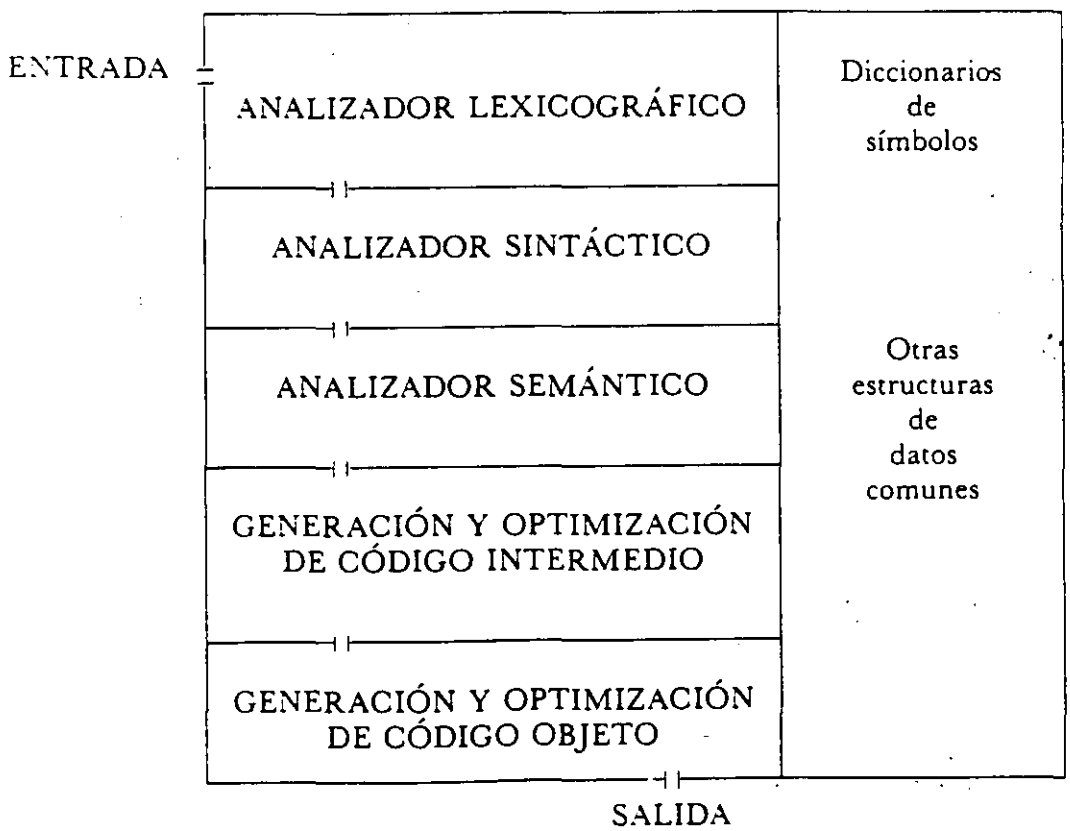
"PROGRAMA" FUENTE	CÓDIGO INTERMEDIO	CÓDIGO OBJETO	
C = A + B	+ (X1, A, B)	CARGA_Ac	A
	= (C, X1, –)	SUMA	B
		GUARDA_Ac	X1
		CARGA_Ac	X1
		GUARDA_Ac	C

Obsérvese cómo el código objeto (comúnmente llamado simplemente código) es de mayor extensión que el programa fuente; esto resulta natural, en

virtud de la correspondencia de uno a varios que existe entre una expresión escrita en un lenguaje de alto nivel y una equivalente escrita en lenguaje de máquina. Nótese también que aún hay redundancias en el código generado. Esto resulta aparente en el cuarto renglón, donde se hace un `CARGA_Ac X1` que sobra, puesto que en el renglón anterior se había hecho la operación contraria y, por tanto, ya no tiene sentido intentar cargar el acumulador con el valor que ya contiene.

Todo compilador, pues, requiere de otra etapa —final— que se conoce como **optimización de código objeto**. Cabe advertir al lector que estos ejemplos mínimos sólo permiten tener una idea vaga del alcance, diversidad y magnitud de los problemas de la generación y optimización de código intermedio y objeto.

La estructura funcional de un compilador aparece resumida en el siguiente diagrama, que interrelaciona sus diversas fases, y las conecta mediante estructuras de datos comunes, que reciben el nombre de diccionarios de símbolos. Estas áreas de memoria guardan información relativa a todos y cada uno de los componentes sintácticos que el compilador ha encontrado a lo largo de sus recorridos lexicográfico y sintáctico del programa fuente, junto con características de los mismos, que serán importantes para la posterior generación de código (por ejemplo, el tamaño de cada variable, su tipo, sus requerimientos de memoria, etcétera).



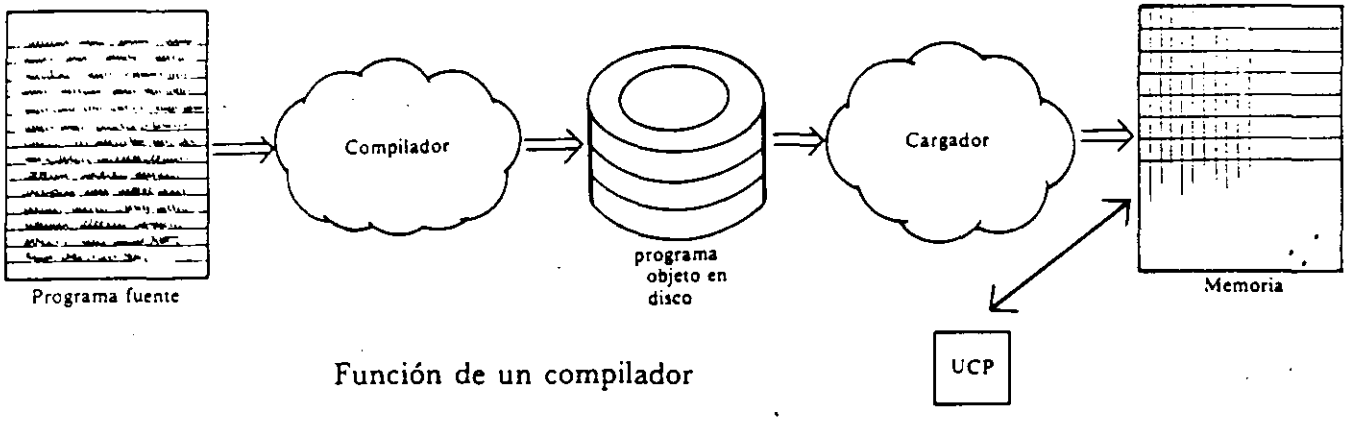
Estructura funcional de un compilador

Cuando el traductor ejecuta inmediatamente el código obtenido recibe el nombre de **intérprete**. Un intérprete no genera código, sino que lo ejecuta tan pronto lo obtiene. Hay una gran diferencia de velocidad entre la ejecución de un programa (objeto) compilado y un programa (fuente) interpretado. La segunda siempre es más lenta que la primera, ya que el intérprete debe analizar, traducir y ejecutar cada instrucción, aun cuando ésta forme parte de un ciclo de repeticiones.

En la actualidad, prácticamente toda la programación de computadoras se hace en lenguajes de alto nivel, por medio de compiladores e intérpretes. La programación en ensamblador se reserva para aplicaciones especiales, que requieren de una optimización cuidadosa, hecha a mano, y para aplicaciones que se relacionan con equipo y hardware especiales.

Existen, además, herramientas de programación especializadas en la construcción de compiladores. Un **generador de analizadores lexicográficos**, por ejemplo, es un programa que recibe como entrada la especificación de la lexicografía de un lenguaje (cómo deben arreglarse sus componentes léxicos o *tokens*), y produce como salida un programa que hace el análisis.

Un **compilador de compiladores** (*compiler compiler*), por otro lado, es un programa que recibe como entrada la gramática de un lenguaje de programación y produce como salida el analizador sintáctico para esa gramática. Como se comprenderá, estos son programas altamente complejos y elaborados.



Función de un compilador

Logramos ya el objetivo especificado al comienzo de este capítulo: hacer una herramienta para comunicarse con la computadora en términos de un lenguaje más cercano al nuestro y menos dependiente con respecto a detalles particulares de la arquitectura de la máquina. Pero esto tiene un precio; comunicarse con la máquina por medio de un compilador añade un paso más a los descritos al final del apartado 4.4, porque ahora es necesario compilar un programa, ensamblarlo, cargarlo y ejecutarlo, en ese orden.

Ahora la comunicación con la computadora requiere tres pasos antes de la ejecución

Todos estos pasos se logran de manera automática por medio de un lenguaje de control, con el cual se especifica a la computadora lo que se desea hacer, para que la máquina tome el control de las operaciones de ahí en adelante. Pero esto es tema de la siguiente sección.

4.6 Sistemas operativos

Cuando en la página 89 se hablaba de un monitor, la referencia era a un programa residente en el sistema de cómputo, que tiene como función controlar los procesos que en él suceden. Ahora que inicia el tema de los sistemas operativos, se estudiarán con más detenimiento los procesos en cuestión.

Como ya se ha visto, el simple hecho de intentar ejecutar un programa escrito en un lenguaje de alto nivel implica la ejecución de varios programas más —que no han sido escritos por el programador, sino que forman parte de la programación de sistemas— como son cargadores, ensambladores, compiladores, editores, etc. Diremos rápidamente que todos estos **programas de utilería*** son coordinados por otro (mucho más grande y complejo), que recibe precisamente el nombre de **sistema operativo**. En este sentido parece no haber gran diferencia entre un monitor y un sistema operativo y, en efecto, ambos parten de la misma idea: que sea la propia computadora la que lleve el control de los procesos, quitando al usuario esa responsabilidad.

La diferencia comienza a ser notoria cuando uno se detiene a pensar en que un procesador común es capaz de ejecutar cientos de miles (e incluso millones) de instrucciones por segundo, y que resulta ridículo dedicar todos esos recursos a una sola persona. Entonces, por ejemplo, ¿qué pasa cuando el programa de aplicación escrito por el programador pide un dato por la pantalla? Sucede que el procesador se detiene a esperar que el usuario digite el valor esperado y, mientras tanto, pueden pasar varios segundos (¡o minutos!). En todo ese tiempo el procesador estará desperdiciando la oportunidad de ejecutar millones y millones de instrucciones, dedicado nada más a esperar. Claramente, esto no está bien. Surge la idea de que durante el intervalo en que el usuario no está aprovechando el procesador, alguien más podría emplearlo. Aquí hay ya una tarea importante para el recién introducido concepto de sistema operativo.

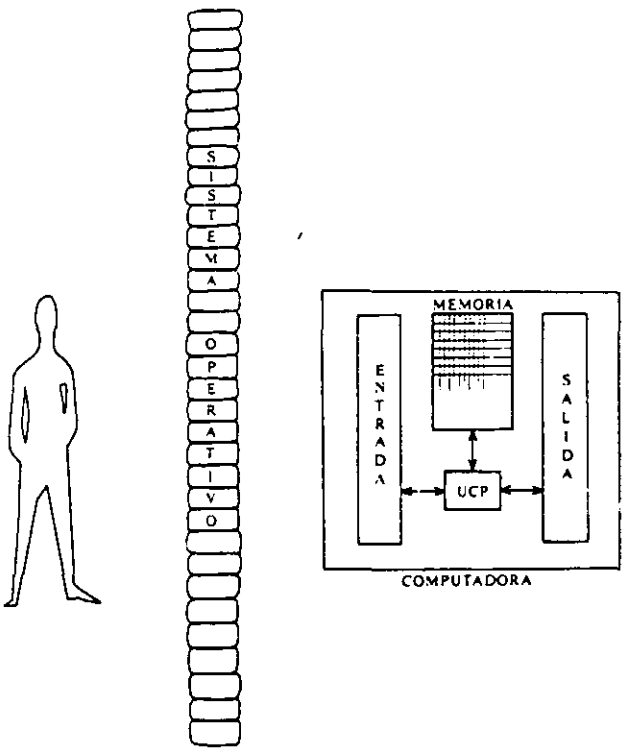
Funciones de un sistema operativo

Otra función que se le puede asignar, y que depende de la anterior, es compartir los recursos de la máquina entre varios procesos al mismo tiempo. Si se considera al procesador como un recurso del sistema de cómputo (al igual que la memoria y los discos magnéticos), salta a la vista cómo la idea de atender a un usuario en los tiempos muertos del procesador entra naturalmente en este esquema, que resulta ser más amplio. De hecho, cuando se estudian estos temas se usan palabras como multiprogramación, tiempo compartido, multiprocesamiento, etc., que están ya permanentemente asociadas al concepto mismo de la computadora moderna. Todos estos conceptos parten, como se dijo, de que el procesador central funciona a una velocidad tal que le permite atender varios procesos.

Del hecho de compartir el procesador sigue un amplio conjunto de tareas afines, tales como compartir la memoria central, el espacio en disco, la impresora, etc. Por tanto, es necesario *administrar eficientemente el sistema de cómputo como un todo armónico*. Y no sólo eso, también es indispensable permitir que

* Reciben el mismo nombre que en el lenguaje del teatro se da a los elementos que sirven para hacer más fácil la labor de montaje de escenografía. Es decir, los programas de utilería se emplean como medio para agilizar el uso normal de una computadora.

los diferentes usuarios se comuniquen entre sí, y protegerlos unos de otros; se requiere, además, permitirles almacenar información durante plazos medianos o largos, y darles la facilidad de utilizar de manera sencilla todos los recursos, facilidades y lenguajes de que dispone la computadora.



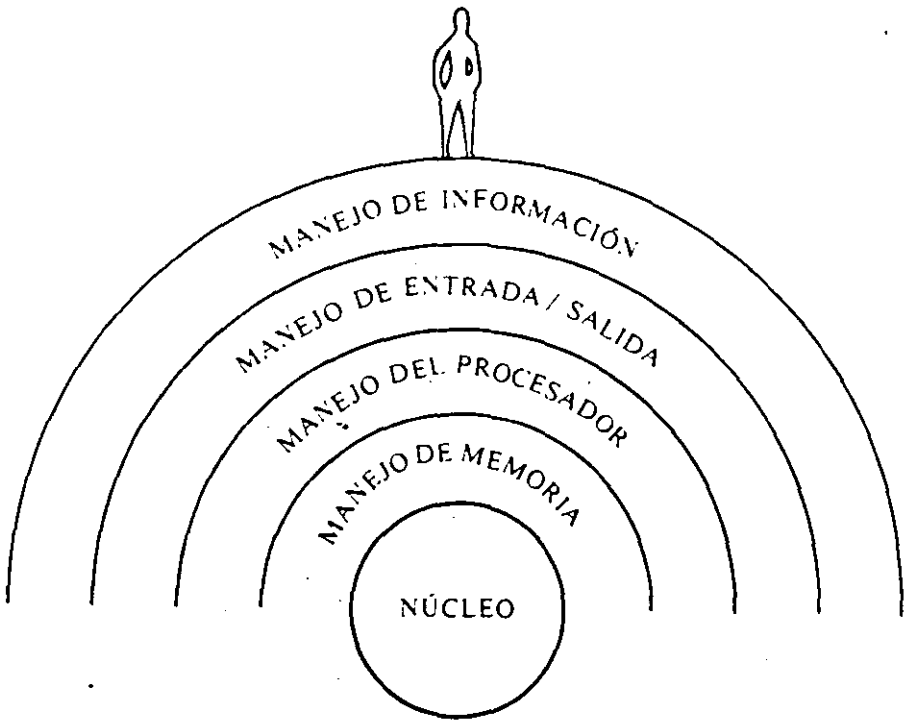
Función de un sistema operativo

Podría resumirse la tarea de un sistema operativo diciendo que su función central es administrar y organizar los recursos de que dispone una computadora para la mejor utilización de la misma, en beneficio del mayor número posible de usuarios.

Modelo de estudio para los sistemas operativos

El estudio de los sistemas operativos suele dividirse en funciones jerárquicas, que van desde niveles muy cercanos a la máquina misma hasta niveles más virtuales, en el sentido de que ya no tratan a la computadora como una máquina (dotada de un procesador y de memoria, etc.), sino como un esquema diseñado para manejar información, sin preocuparse demasiado por detalles como registros, bloques, etcétera.

Esta es una idea fundamental; un sistema operativo convierte a una máquina computadora "real" en una computadora "virtual", que es capaz de hacer cosas cualitativamente diferentes a las de su contraparte física. Para entender esto, piénsese en el concepto usual de lo que es una computadora,



Un modelo de estudio para los sistemas operativos

y se apreciará que, en general, uno se refiere a una computadora en términos virtuales, y no reales ("una computadora es un cerebro eléctrónico capaz de almacenar muchísima información y manejarla y procesarla a enorme velocidad", o "es la encargada de cobrarme el teléfono").

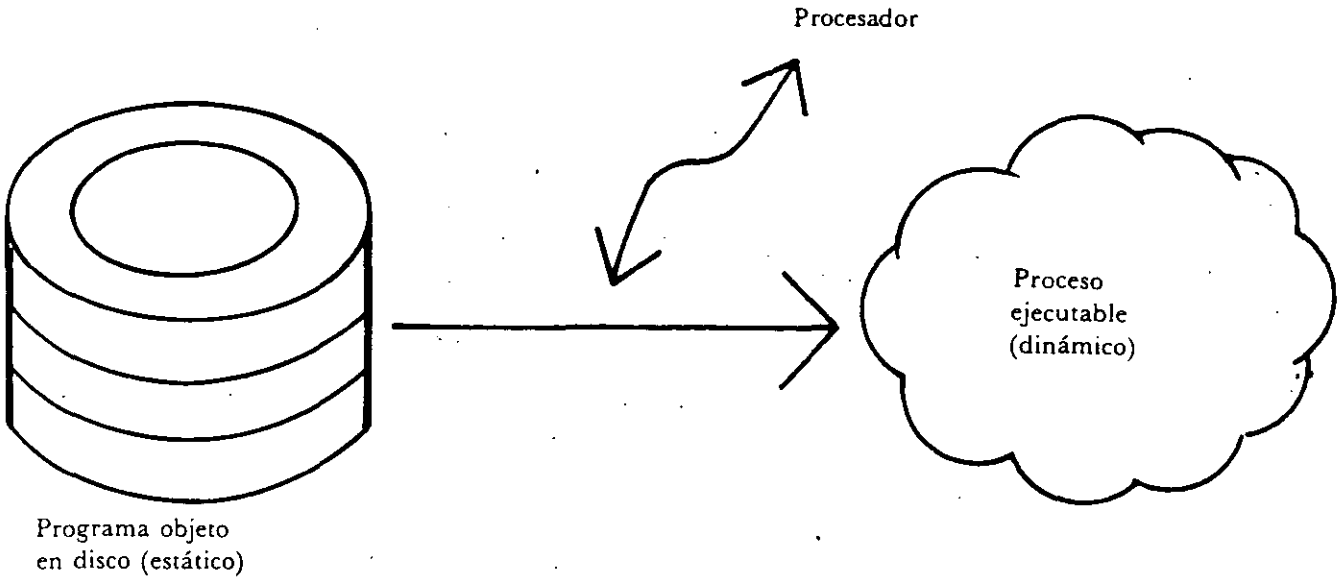
Si se piensa en una máquina por un lado y en el ser humano por el otro, inmediatamente saltará a la vista que existe un abismo en sus capacidades de comunicación; el lenguaje que la computadora maneja no es más que una burda imitación del lenguaje que hablamos los humanos y, por tanto, la comunicación que se puede establecer entre máquina y hombre es muy rudimentaria. La función general de la programación de sistemas consiste en hacer más fácil el camino que nos separa de las computadoras, y la de los sistemas operativos en particular consiste en lograr que la comunicación se haga de manera tal que el humano vea una imagen virtual de la computadora, y no necesariamente note que lo que tiene enfrente es un aparato dotado de un acumulador y medio millón de celdas de memoria.

Esta no es una tarea fácil, pues se requiere la automatización de cientos (o miles) de pequeñas tareas, que van de lo trivial a lo enormemente complejo. La jerarquización mencionada parte del hecho de que un sistema de cómputo está configurado (como se vio en los capítulos anteriores) alrededor del procesador y de la memoria, y a partir de estos recursos se van formando herramientas de programación de sistemas (esto es, programas específicos) que permiten utilizar la máquina como sistema, y no como partes aisladas.

El esquema que suele seguirse para el estudio de los sistemas operativos recibe el nombre de "modelo de la cebolla", debido a que está formado por

capas concéntricas alrededor de un núcleo. La parte interna del conjunto jerárquico de programas que forman un sistema operativo recibe el nombre de **núcleo** (o *kernel*, en inglés). Algunas de las otras capas se encargan del manejo de la memoria, el procesador, los dispositivos de entrada/salida, los archivos, etcétera.

Para el análisis que sigue es necesario definir algunos términos. Se llama **programa** a un *conjunto de instrucciones* escritas en algún lenguaje de computación (en este momento no importa si están en un lenguaje de alto nivel o si consisten en cadenas de unos y ceros del lenguaje de máquina). La característica principal de un programa, en este sentido, es que es la especificación de un conjunto de instrucciones estáticas, puesto que están escritas y aún no se han ejecutado. Sólo son un adelanto de lo que va a suceder cuando se les "dé vida". Claro que cuando un programa se ejecuta, lo que se observa ya no es un conjunto de instrucciones, sino uno de acciones, que no son otra cosa que las instrucciones en estado activo. Se llama **proceso** a un *conjunto de acciones* —dinámicas— que son el resultado de la ejecución de un programa. Es decir, el concepto de programa es anterior necesariamente al de proceso. El agente que "da vida" a una instrucción (para convertirla en acción) se llama **procesador**



Programa vs. proceso

Se dice que dos procesos son **concurrentes** cuando se ejecutan (acción por acción) en el mismo *intervalo* de tiempo; y dos procesos se consideran **simultáneos** cuando se ejecutan en el mismo *instante*. Es decir, para que exista simultaneidad entre n procesos, se debe forzosamente contar con n procesadores, mientras que para ejecutarlos concurrentemente se requiere tan sólo "repartir" el procesador entre ellos a una velocidad tal que, por unidad de tiempo

(o intervalo), todos reciban su atención (aunque sea parcial). Este último concepto —que es la base de la multiprogramación y el tiempo compartido— recibe el nombre de **multiplexación en tiempo**

El núcleo del sistema operativo

Está claro que el problema de la concurrencia entre procesos tiene que resolverse en el nivel más cercano posible al núcleo del sistema operativo, ya que la multiplexación del procesador es una operación primitiva, es decir, combinándola con otras, sirve para formar funciones más complejas dentro del sistema.

Las funciones de un núcleo consisten en tomar el control del procesador y determinar cuándo y cómo lo va a repartir entre diversos usuarios.

En términos generales sucede que el procesador abandona el proceso que está siendo ejecutado, y dedica su atención a ejecutar otro, cuando el primero entra en algún estado de espera. Como se dijo antes, un proceso entra en estado de espera cuando pide efectuar alguna operación que sea muy lenta en comparación con las velocidades normales de procesamiento. Las operaciones lentas casi siempre son las de entrada/salida de datos de la computadora desde/hacia el mundo exterior. Surge así una primera pregunta: ¿cómo detecta el sistema operativo que se trata de una operación lenta?

Manejo de interrupciones

Este problema se resuelve clasificando el conjunto de operaciones que puede efectuar un procesador como "normales" o "privilegiadas". Por definición, una operación privilegiada es aquella que, al ser ejecutada, causa que el procesador entre en un estado especial llamado interrupción (véase la pág. 44).

Durante la interrupción, el procesador se detiene momentáneamente y pregunta si puede (o debe) seguir ejecutando. Esta pregunta consiste en que el procesador ejecute automáticamente un pequeño programa de atención a la interrupción que averigua la causa de ella y determina los pasos a seguir.

Un ejemplo aclarará esto. Supóngase que un proceso pide la ejecución de una operación (privilegiada) de entrada/salida. El procesador está diseñado, de tal forma que en ese momento se produce una interrupción. Instantáneamente ocurre un desvío en la secuencia de la ejecución de instrucciones, y en vez de continuar ejecutando el programa, el procesador comienza a ejecutar una rutina —previamente codificada por el diseñador del sistema operativo, y residente en un área fija y preestablecida de la memoria— que atiende la interrupción. Este programa del sistema averigua la causa del desvío y determina, para este caso, que se intentó ejecutar una operación privilegiada de entrada/salida. Asimismo, este programa indica al procesador lo que debe hacer a continuación.

Toca entonces determinar cómo aprovechar el procesador mientras el proceso original es atendido por algún dispositivo de entrada o salida. Esto depende, naturalmente, de si hay o no otros procesos participando en la multiplexación de la unidad central de procesamiento (compitiendo por el procesador).

Si no hay otro proceso en espera del procesador, tan sólo se devuelve el control al proceso original, y el procesador espera pacientemente —desperdiciando cientos de miles de ciclos de máquina— a que se complete la operación de entrada/salida deseada. Esto recibe en inglés el nombre de *idle-wait*, (espera ociosa).

Pero si existen más procesos en estado de espera, entonces sucede algo muy interesante; el proceso original se "congela" —luego veremos cómo— y se deja en estado de espera, en tanto que otro proceso se "descongela" y recibe el control sobre el procesador. Ahora la operación del sistema de cómputo sigue como antes, ¡pero ejecutando un proceso diferente!

Por supuesto que en una computadora real todos estos pasos no toman más que algunas milésimas de segundo.

Surgen muchas preguntas. ¿Cómo se representa un proceso dentro de una computadora? ¿Cómo se activa y desactiva un proceso? ¿Cómo se escoge el proceso que se activará ahora? ¿Quién y cómo decide el conjunto de operaciones necesarias para lograr todo esto? Se intentará responder a estas preguntas conforme se defina en qué consiste el núcleo de un sistema operativo.

Composición de un núcleo

El núcleo de un sistema operativo está formado, en términos generales, por tres subsistemas. El primero se encarga de manejar las interrupciones del procesador central, de la manera ya descrita. El segundo tiene como función escoger (y activar) un nuevo proceso para ser ejecutado, y la operación inversa ("congelar" el que fue interrumpido). El tercer programa cumple una función muy importante: coordinar los diversos procesos (del sistema operativo y de los usuarios) que interactúan en el núcleo del sistema operativo para que no choquen entre sí, es decir, para que la UCP no se confunda. Esta última función es tema de un estudio más detallado, que queda fuera del alcance de este curso.

Las tres funciones son desempeñadas por otros tantos procesos del núcleo que reciben los nombres de **manejador de interrupciones de bajo nivel, despachador** y **semáforos**. En las referencias [BENA82], [BACM86], [BRIH73], [MADS74], [PETJ83] y [TANA87] se pueden encontrar capítulos completos que describen todo esto con mucho más detalle.

El despachador "congela" un proceso almacenando —en registros especiales de la UCP— los datos volátiles que resultaron de su ejecución, hasta antes de ser desactivado. Por ejemplo, si se almacenan los contenidos de los diferentes registros de trabajo, del acumulador, y de otros elementos del procesador central que intervienen en un cálculo, el proceso puede ser desconectado del procesador, sin que se pierda el avance de lo calculado hasta ese momento. Para esto sirven operaciones como las descritas en la pág. 78, dentro del "grupo de control del sistema" del lenguaje de máquina.

Cuando proceda, el despachador reactivará ese proceso simplemente copiando los contenidos de esos registros especiales de regreso en el acumulador y demás registros de trabajo de la UCP, de modo que el proceso recién despertado pueda continuar su ejecución, como si nada hubiera sucedido.

Los registros especiales donde el despachador almacena la información volátil de un proceso que se desactivará reciben el nombre (establecido por IBM en la segunda generación) de *program status word* (**PSW**) o **vector de estado**

No hay que confundir, sin embargo, el hecho de almacenar la información volátil de un programa con el hecho —que no sucede— de almacenar el contenido de las celdas de memoria que está utilizando cierto proceso.

Está claro que esto último no tendría sentido, pues para almacenar la información contenida en, digamos, diez mil celdas de memoria, se requieren, precisamente, diez mil celdas adicionales, lo cual resulta absurdo. Es obvio que se deben separar esas diez mil celdas (que pertenecen a un proceso en particular) y no permitir que sean utilizadas por ningún otro, so pena de perder la información original.

Esto lleva naturalmente a la segunda función del sistema operativo, que es controlar el uso de la memoria. Se explorarán diversas ideas y enfoques que existen para tratarla.

Manejo de memoria

Considérense los problemas que enfrenta el sistema de cómputo para dar atención concurrentemente a seis usuarios. Por un lado, tiene que multiplexar el procesador entre sus seis clientes y, a la vez, dar el control a las rutinas del sistema que se encargan de muchas tareas auxiliares (desactivar/activar procesos, mandar mensajes, controlar las operaciones globales, etc.). Debe determinar un orden óptimo de multiplexación (que se analiza más adelante), y tiene que subdividir la memoria en seis porciones, una para cada proceso activo. Un proceso residente en memoria se llama activo cuando se encuentra ya sea en estado de ejecución o en estado de desactivación (o sea, no está en ejecución en ese momento, pero puede estarlo en cualquier otro momento cuando sea "descongelado" por el despachador). Un proceso cualquiera puede estar en uno de varios **estados**: activo, en ejecución, o residente en disco magnético. La función del manejador de la memoria consiste en mantener un espacio en ésta para todos los procesos activos dentro del sistema.

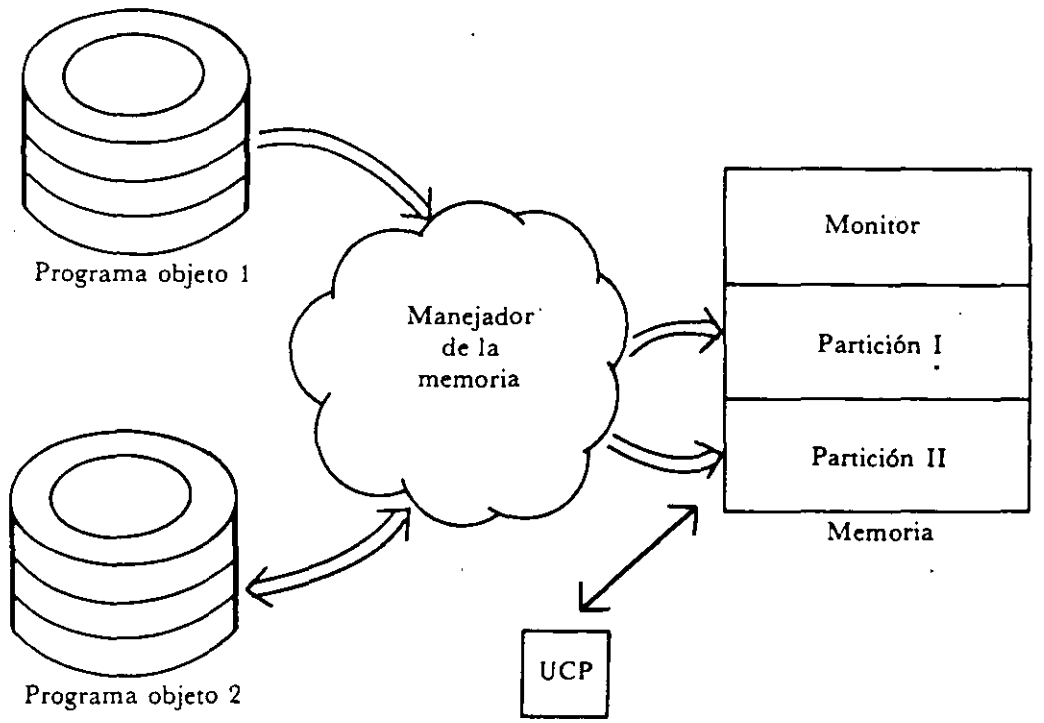
Primer esquema de manejo de memoria: memoria única asignada

Existen varias maneras de manejar la memoria en un sistema de cómputo. La más sencilla es asignar toda la memoria disponible a un solo usuario, pero esto no permite más que un usuario en operación. El primer esquema para permitir la **multiprogramación** (o sea, la activación de varios procesos simultáneamente) recibe el nombre de **manejo de memoria por particiones**, y consiste en subdividir la memoria en varias secciones fijas y asignar cada una de ellas a un usuario o proceso activo. El principal problema por resolver es asegurar que ningún usuario intervenga en el área de memoria asignada a otro. Desde este punto de vista, el manejo de memoria consiste en controlar cuáles particiones están asignadas a cuáles procesos, para poder liberar particiones cuando los procesos residentes en ellas terminen o cambien, y poder ofrecer particiones libres a procesos que soliciten atención por parte del sistema de cómputo.

Segundo esquema de manejo de memoria: por particiones

La ventaja fundamental de este modelo es que permite la multiprogramación, y su principal desventaja consiste en que deja lugares libres en la memoria que, como son de tamaño fijo, no pueden ser utilizados más que por procesos de longitud menor o igual a la de la partición en cuestión. Sucede

muchas veces que, por ejemplo, aun cuando hay dos particiones libres de diez mil celdas cada una, no se puede dar atención a un proceso (que se encuentra en estado de espera en el disco magnético) que mide doce mil celdas de longitud. La razón técnica de esto es que no hay una partición de tamaño suficiente en este momento, aunque sí exista área suficiente en la memoria; lo que sucede es que está particionada. Este problema, llamado **fragmentación externa**, se podría evitar permitiendo que una partición pueda fusionarse con otra, para lograr una partición nueva de más capacidad.



Manejo de memoria por particiones

Tercer esquema de manejo de memoria: particiones relocalizables

Esto da lugar a un nuevo esquema de manejo de memoria que recibe el nombre de **particiones relocalizables**. La idea consiste en mover celdas de memoria de un lugar a otro para juntar las áreas libres en un mismo lugar. Las celdas no se mueven, sino que sus contenidos se copian de un lugar a otro, y aunque con esto se crea un nuevo problema —el de la relocalización—, permite mayor flexibilidad que el anterior; sólo que resulta más costoso, puesto que hay que compactar (mover) los procesos a tiempo de ejecución y realizar algunos cambios en el procesador central, para evitar que este desplazamiento cause problemas con respecto a las direcciones. Sucede que si un proceso estaba cargado a partir de la celda 1000, las direcciones absolutas de su espacio de direcciones tienen esta celda como origen. Pero si se relocaliza y se coloca a partir de la celda 3478, por ejemplo, las referencias a todas las direcciones tienen que alterarse —sumándoles el desplazamiento de 2478 celdas—, a fin de que el proceso pueda continuar su ejecución como antes. El procesador central se encarga de este ajuste a tiempo de ejecución

por medio de un componente electrónico adicional que se conoce como **registro de relocalización**

Cuarto esquema de manejo de memoria: *swapping*

En algunos sistemas se recurre al sencillo expediente de quitar por completo de la memoria un proceso que está desactivado, copiándolo al disco magnético y liberando así un área significativa en la memoria central. Cuando llegue el momento de volverlo a ejecutar se cargará nuevamente trayéndolo del disco magnético en el que reside. Este esquema recibe el nombre de ***swapping*** (intercambio).

Quinto esquema: paginación

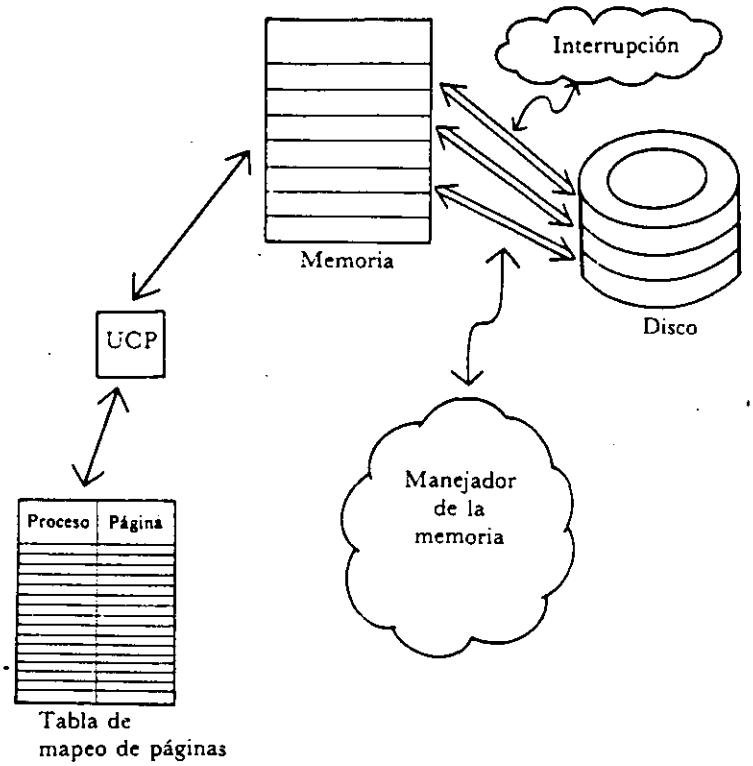
Debido a los costos que representa la compactación (ya que es necesario detener la ejecución del proceso para efectuarla) o el *swapping* (porque el traslado hacia/desde el disco magnético toma tiempo), se inventó otro esquema, más ágil y eficiente, llamado **paginación**. Esto consiste en dividir los procesos en fragmentos de longitud fija, llamados páginas, que se almacenan en áreas de igual tamaño en memoria, llamadas bloques. Esto es, cada página de cada proceso se guarda en un bloque en memoria. Un proceso común puede constar de quince páginas, residentes en memoria en otros tantos bloques. La ventaja radica en que no es necesario que las páginas de un proceso estén contiguas en la memoria, quedando automáticamente eliminado el problema de la fragmentación externa. Con la ayuda de una **tabla de mapeo de páginas**, que controla cuáles páginas de qué procesos residen en cuáles bloques de memoria, se puede implantar un esquema muy ágil de manejo de memoria central, controlado por el sistema operativo.

De acuerdo con lo dicho, si ya no es necesario que todas las páginas de un proceso estén cargadas de forma contigua en la memoria (gracias a la tabla de mapeo), entonces tampoco hay necesidad de que todas las páginas de un proceso determinado estén residentes (contiguas o no) en memoria. Es decir, se podría comenzar a ejecutar un proceso cuando tan sólo una parte del mismo esté cargada en memoria, e ir cargando a tiempo de ejecución las páginas que se requieran. Esta importante idea recibe el nombre de **memoria virtual** (cf. pág. 58, y es la base sobre la cual descansa el enorme poderío de una computadora grande, y la razón por la que una máquina puede atender a muchos usuarios al mismo tiempo aunque disponga de una memoria limitada.

Cuando un proceso pide una página no residente en la memoria, el sistema operativo lo detecta por medio de una interrupción, que es atendida por el manejador de interrupciones del núcleo. Éste determina la causa (interrupción por página) y copia la información solicitada —residente en el disco magnético— en un bloque libre de la memoria.

Sexto esquema: paginación por demanda, que además ofrece memoria virtual.

Este nuevo esquema de manejo de memoria se llama **paginación por demanda**. Sus ventajas son obvias, pues permite una tremenda flexibilidad en el uso de los recursos del sistema. Su desventaja es, fundamentalmente, su enorme complejidad. En efecto, los sistemas operativos de este tipo constan de decenas de miles de instrucciones, y son escritos por grupos enteros de programadores durante meses y meses, además de que se requiere un considerable auxilio por parte de los circuitos electrónicos para que la velocidad de procesamiento no disminuya radicalmente por la gigantesca cantidad de operaciones adicionales que el sistema debe ejecutar. Como la tabla de



Memoria virtual

páginas reside en la memoria central, y es necesario consultarla para cada acceso, se requiere un ciclo de lectura de memoria adicional (ciclo de *fetch*) por cada operación sobre una página, lo cual claramente es inaceptable en términos de la reducción de velocidad de proceso resultante. Por tanto, los sistemas de paginación por demanda emplean mecanismos adicionales de hardware para auxiliarse en esta tarea. Uno de ellos es el conocido como memoria *caché* o memoria auxiliar rápida, en la que se guardan los contenidos activos de la sección de la tabla de páginas en uso, reduciendo grandemente el tiempo extra requerido por cada consulta. Muchos procesadores recientes trabajan en colaboración con otro complejo subsistema electrónico para el manejo de estas tareas de paginación, que recibe el nombre de **unidad de manejo de memoria** (MMU: *memory management unit*).

Séptimo esquema de manejo de memoria: segmentación

Existe otro esquema de manejo de memoria que también permite memoria virtual, y se llama manejo de memoria por **segmentación**. En éste, los procesos se dividen en fracciones llamadas segmentos. Un segmento es una unidad lógica autocontenida (un programa completo, una subrutina o un área grande de datos) que se carga en forma independiente en la memoria. La diferencia con respecto a las páginas es que aquéllas son de longitud fija, mientras que los segmentos son variables, dependiendo de la cantidad de código que contenga el programa o subprograma que representan. El manejo de segmentos es parecido al de páginas, aunque tiene ciertas ventajas sobre éste, que ya no se estudian por tratarse de un tema especializado. Basta con saber que una máquina con sistema operativo de segmentación es por lo me-

nos tan poderosa y compleja como otra que maneje memoria virtual por paginación.

Último esquema de manejo de memoria

Un último esquema —el más complejo de todos— combina las ventajas de la paginación por demanda con las de la segmentación, y recibe el nombre de **segmentación-paginación**, pero sólo lo utilizan algunas computadoras realmente grandes.

Como se ha visto, el problema del manejo de memoria es extenso y complicado, pero es fundamental para comprender lo que es un sistema operativo. Sin embargo, apenas se han estudiado dos capas del modelo de la cebolla: el núcleo y el manejador de memoria.

Lo que sigue es determinar en qué orden y con qué criterios se dará atención a los diversos usuarios de un sistema de cómputo o, en otras palabras, quién determina cuáles procesos estarán activos y cuándo. Esta es función del siguiente nivel.

Manejo del procesador

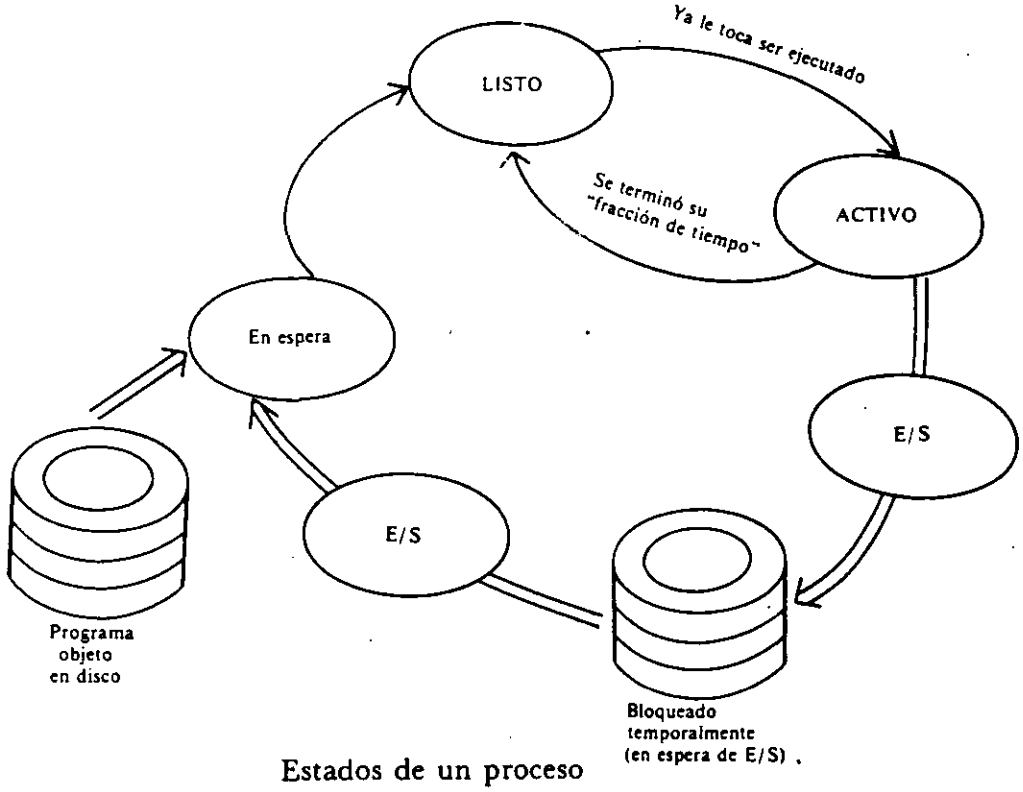
Este programa del sistema (a veces conocido como **despachador de alto nivel** o *scheduler*) se encarga de determinar el orden óptimo de atención a los diversos procesos que están compitiendo por ganar la atención del procesador central. Su principal característica es la capacidad de afrontar la indeterminación, es decir, el desconocimiento del orden en que se van a presentar los diversos procesos, el número de ellos y su composición. Está claro que, empleando un término de la filosofía existencialista, una computadora simplemente "está ahí" y no se puede predecir el uso que darán al procesador los numerosos procesos que existen, por la sencilla razón de que no existe un determinado plan de acción, pues los procesos son independientes unos de otros y pertenecen a usuarios que no se conocen entre sí.

El factor de indeterminación hace que estos programas que manejan el procesador sean complicados y deban considerar criterios estadísticos y suposiciones acerca del comportamiento de los posibles usuarios. Esto obliga, por otro lado, a que el sistema maneje un conjunto de colas de espera de los procesos que no puede (o no debe) atender en un cierto momento. Estas colas son áreas del disco magnético donde se almacenan los programas que desean ingresar al sistema de cómputo.

Convertir un programa en un proceso

Tal vez la función más importante del *scheduler* sea convertir los programas de los usuarios en procesos para el sistema. Es decir, en tomar los programas originales y asignarles una representación interna que permita que el sistema operativo determine los recursos que los procesos requieran de la computadora. Por ejemplo, es necesario averiguar, con un mínimo de precisión, cuántas celdas de memoria requerirá un proceso cuando entre en ejecución, qué área de disco necesitará, cuánto tiempo de procesador central espera consumir, etc. Todo esto con la finalidad de que el sistema operativo pueda realizar una planeación eficiente de la distribución de los recursos de cómputo entre los diversos (y aún no conocidos) usuarios.

A manera de ejemplo, supóngase que en un supermercado dos clientes van a hacer cola en la caja de salida, uno con muchas mercancías para pagar y otro con sólo un artículo, pero no es posible determinar inicialmente a cuál se atenderá primero porque esto depende de su orden —indeterminista— de llegada a la caja. Suponiendo que ambos llegaran simultáneamente a la caja y que para atender a uno fueran necesarios diez minutos —porque lleva muchas mercancías— y el otro fuera atendido en un minuto, está claro que conviene atender primero a este último, porque no es grave que una espera prevista de diez minutos se extienda a once, pero, por supuesto, hacer que una de un minuto tarde once, sí constituye una carga para ese usuario. Naturalmente, el cliente menor quedará más satisfecho con el sistema si es atendido con prontitud, lo cual depende de la programación que se pueda hacer cuando ambos estén listos para llegar a la caja de salida.



Estados de un proceso

Esto implica también la necesidad de que exista un proceso del sistema operativo que se encargue precisamente de averiguar cuáles procesos ya terminaron de ejecutarse, cuáles están listos para comenzar a hacerlo, cuáles van a imprimir sus resultados, etc. Este proceso recibe el nombre de **controlador de tráfico**.

El manejador del procesador de bajo nivel (que reside en el núcleo, y que ya se describió) puede estar guiado por diversos criterios, que determinan el tipo de operación de la computadora. Cuando las interrupciones se deben a que los procesos piden la ejecución de operaciones de entrada/salida, se dice

que el sistema es de multiprogramación, como se ha visto. Pero cuando se decide atender cada proceso durante un tiempo fijo (un segundo, por ejemplo), entonces se está hablando de un sistema de tiempo compartido. No hay una distinción tajante entre la multiprogramación y el **tiempo compartido**, y a veces el primer concepto incluye al segundo, aunque estrictamente hablando sí hay diferencias entre ambos. Lo importante es que estos dos esquemas son el motor del manejador del procesador de alto nivel que recién se describió. En el artículo [SCIA71] se puede encontrar una descripción del origen de estas ideas.

Una vez estudiadas las funciones mínimas de los procesos, queda aún por resolver el problema de la comunicación entre los procesos que están en ejecución y el mundo exterior. Para un proceso, el mundo exterior se refiere a las unidades de entrada/salida de la computadora y, de manera adicional, a las unidades de memoria secundaria. La siguiente capa del sistema operativo maneja las funciones de entrada/salida, o sea, la comunicación de los procesos con su entorno.

Manejo de entradas y salidas

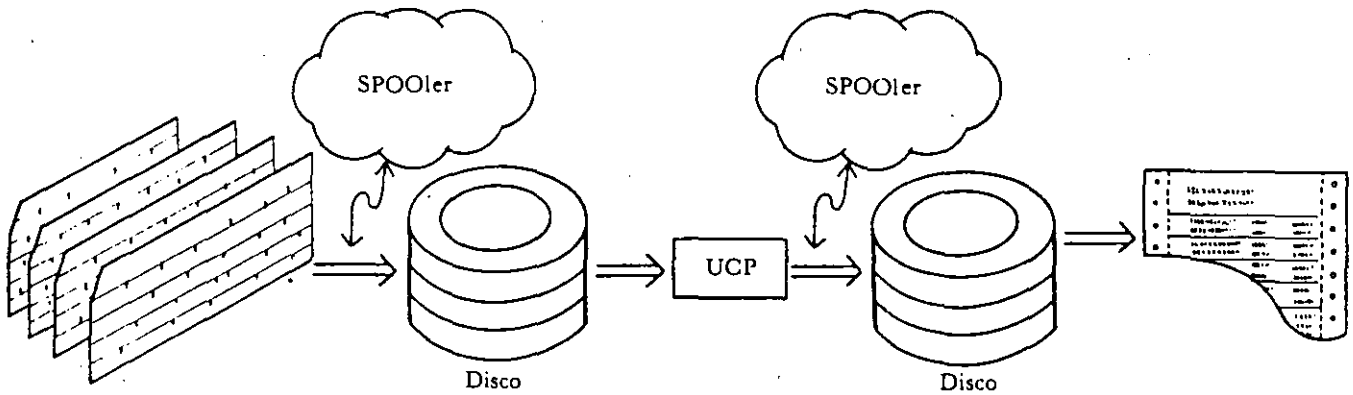
El manejador de entrada/salida tiene como función principal atender los pedidos que los procesos en ejecución hacen sobre las unidades periféricas. Esta atención requiere, la mayoría de las veces, una traducción lógica y física entre las diversas unidades involucradas. La parte física hace que puedan comunicarse aparatos diferentes entre sí aunque manejen códigos internos distintos, y la traducción lógica —más interesante para nosotros en este momento— tiene como función virtualizar los pedidos de entrada/salida y postergar su ejecución física lo más posible. En el anexo 3.7 se describió un modelo general (conocido como el modelo ISO/OSI) para la intercomunicación entre diversas computadoras que se encarga, entre otras cosas, de la resolución de las diferencias de códigos entre equipos conectados en una red.

La aplicación típica del concepto de manejo de entrada/salida hace que esos pedidos se virtualicen y sean desviados del/al disco magnético, para que no dependan de las limitaciones de los aparatos físicos de lectura o escritura. Cuando un proceso en ejecución manda un carácter a la impresora, el sistema operativo lo envía al disco magnético, a un área especial destinada a ser la impresora virtual para ese proceso. Y dado que se está hablando de multiprogramación, también se habla de tantas impresoras virtuales, en disco magnético, como procesos activos haya en el sistema.

Lo mismo cabe señalar para el caso de las lecturas. En muchas grandes computadoras primero se leen los datos y se guardan en una lectora virtual en disco magnético para que, cuando el proceso pida un dato, éste le llegue del disco y no de la unidad física de lectura.

Este concepto recibe el nombre de ***SPOOLing*** (*simultaneous peripheral operations on line*, operación simultánea de periféricos en línea). Sus ventajas son claras; al permitir una virtualización de las unidades de entrada/salida de la computadora, ésta se comporta como si tuviera varias de cada una y los pro-

cesos no tienen que esperar a que la impresora esté libre para seguir ejecutando. Además, permite redirigir los archivos de impresión hacia la primera impresora desocupada, para el caso de que la computadora disponga de varias; asimismo, posibilita la reimpresión de múltiples copias del mismo resultado de un programa, grabado previamente en el disco. En los enormes centros de cómputo en los que se procesa, por ejemplo, la nómina de los empleados federales, no deja de resultar impresionante cómo una gran computadora imprime miles y miles de cheques u otros documentos en una decena de impresoras de gran velocidad en forma simultánea.



SPOOLing

Esta capa se encarga también de las transferencias físicas de información entre las unidades de entrada/salida y los procesos en ejecución. Como ya se dijo, realiza en forma automática las traducciones necesarias entre códigos diversos y velocidades de acceso diferentes. Una sección muy importante de ella es el subsistema de manejo lógico del disco magnético, que virtualiza los pedidos de información, para que los procesos no tengan que preocuparse de cuál pista o sector del disco contiene el dato pedido. El proceso en ejecución simplemente solicita el valor de la variable ALFA que está residente en disco, en el área dedicada de antemano por el sistema, y entonces se traduce este pedido al disco magnético como "mover el brazo de lectura del disco al sector 15 de la pista 256, para leer de ahí la información que contenga y colocarla en un área de almacenamiento temporal". Una vez hecho esto, el sistema le avisará al proceso (por medio de una interrupción) que ya puede recoger, de dicha área, los datos que se habían solicitado.

Manejo de información

Una vez resueltos los problemas de lograr el acceso a la información de manera física, queda el de hacer uso "humano" de ésta. De eso se encarga

una nueva capa del sistema operativo, que recibe el nombre de **sistema de archivos** o (manejador de información).

Sus funciones son permitir a los usuarios el manejo libre y simbólico de prácticamente cualquier cantidad de información que deseen almacenar, leer, imprimir, alterar o desechar. Se trata de un manejo libre porque en lo posible tendrá el menor número de restricciones físicas o lógicas, y simbólico ya que el usuario no tendrá que preocuparse de los modos de acceso al disco magnético ni de otros detalles, que ya han sido virtualizados por el manejador de entradas/salidas, y simplemente hará referencia a su información por el nombre simbólico que decidió asignarle libremente.

No sólo eso, sino que el sistema también puede almacenar la información por plazos indefinidos, y recuperarla en cualquier momento, a la vez que maneja criterios de seguridad de acceso y de protección. Todo sistema operativo moderno garantiza de alguna manera la seguridad e integridad de la información que le ha sido confiada por los usuarios, y lo logra por diferentes medios, que van desde mantener copias ocultas hasta revisar periódicamente que se respalde de forma automática en algún lugar seguro. Hay sistemas que incluso reconstruyen archivos dañados por accidente o negligencia.

A estas alturas no será una sorpresa para el lector enterarse de que los sistemas operativos realmente grandes constituyen verdaderos esfuerzos de ingeniería humana y de programación, de una complejidad tal que no existen personas que por sí solas comprendan en detalle su operación completa, sino que son producto de equipos de ingenieros, analistas y programadores.

Aunque nos hemos extendido en la descripción de un sistema operativo, quedan aún por explorar muchos conceptos, a los que se dedica la siguiente sección, y que sirven como apoyo a la programación; se trata de utilerías controladas por el sistema (editores, paquetes diversos, hojas de cálculo, bases de datos, etc.). También se trata, en otra sección de este capítulo, de la inteligencia artificial, área que está adquiriendo tal relevancia que las computadoras de la siguiente generación estarán basadas en ella.

La comunicación final con el usuario

Por último, es necesario que todo sistema operativo se comunique con los usuarios de alguna manera. Y existen básicamente dos formas de lograrlo: mediante un **lenguaje de control** que el interesado aprende, o por medio de menús* que el sistema despliega en la pantalla, para que el usuario escoja la operación que desea efectuar. Ambos sistemas tienen ventajas y desventajas, aunque son más los sistemas operativos que manejan el concepto de lenguaje de control que el de menús.

Por lo que se ha visto, una computadora (en el sentido amplio) no es tal sin un sistema operativo que le dé soporte y la haga aparecer como mucho más de lo que realmente es: un complejo aparato electrónico. De aquí en adelante cuando se hable de un equipo de cómputo se considerará, sin falta, al sistema operativo como integrante indispensable del mismo, no obstante que ya sabemos que se trata de un conjunto de procesos de la programación de

* Este concepto, que ha adquirido popularidad, muestra al usuario una lista de posibilidades para que éste escoja la que requiera, evitándole la necesidad de memorizar las órdenes específicas. Algunas computadoras extienden esto e incorporan teclas especiales en las terminales de video para tener acceso directo a las funciones del menú, lo cual vuelve la operación aun más fácil y conveniente.

sistemas que le da "vida" y potencia a la gran máquina. Para mayor información de carácter general se recomiendan los libros [DEIH83] y [LISM85], y los artículos [DENP71] y [DENP84], además de los ya mencionados en la sección dedicada al núcleo del sistema operativo.

Los siguientes capítulos del libro se dedicarán a explorar y aprender las técnicas de programación que hacen posible escribir sistemas que van desde los muy sencillos hasta los enormemente complejos, como los que se han descrito.

4.7 Utilerías: editores, bases de datos, hojas de cálculo

Como se mencionó en el capítulo 1, las computadoras personales han vuelto notoria la existencia de un conjunto de programas y sistemas utilitarios que, si bien ya existían en las computadoras de la tercera generación, han mejorado y se han vuelto virtualmente indispensables en la operación usual de las computadoras actuales, debido a que presentan muchas ventajas de uso y permiten que la máquina sea empleada eficazmente por no especialistas en computación y, por ende, realmente cumplen la función de puente que comunica a los usuarios con la computadora en una forma práctica y conveniente.

En esta sección se describirá conceptualmente qué es y cómo funciona cada uno de estos sistemas, pero debe quedar claro que no se tratarán cuestiones operativas ni de detalle. En [GRAR88] se puede encontrar una descripción general de algunos de los paquetes que han tenido más éxito en el entorno de las microcomputadoras.

Editores

En el ejemplo dado en el capítulo 3, donde se describía una sesión típica en un centro de cómputo, se mencionó que un editor es un programa del sistema que sirve para introducir textos en la computadora mediante la terminal de video, así que se partirá de que ésta es su función primordial: servir de canal de entrada de textos, datos y programas fuente, que normalmente residirán en el disco magnético de la computadora hasta que se decida borrarlos o transferirlos a una cinta o diskette de respaldo.

En el diseño de un editor existen dos frentes que hay que considerar: la comunicación con el usuario y la liga con el sistema de archivos de la máquina. Los objetivos de la primera interfaz son permitir que el usuario pueda crear, manipular y borrar información en forma fácil y sin tener que preocuparse de los detalles físicos de operación de la memoria o los discos, mientras que los de la segunda función consisten en lograr que la gran cantidad de operaciones lógicas y físicas requeridas para interactuar con los sistemas de archivos y de entrada/salida de la computadora se lleve a cabo rápida

y eficazmente, consumiendo la menor cantidad de recursos posible para que el tiempo de respuesta no se degrade.

Es decir, un editor debe ser capaz de traducir a la computadora las órdenes que el usuario le da, y que normalmente se clasifican en alguna de las categorías que a continuación se mencionan. Cada vez es menos clara la distinción entre un editor y un procesador de palabras (como los descritos en el capítulo 3), por lo cual es posible que algunas de las funciones aquí mencionadas estén incluidas en varios de los más poderosos procesadores de palabras comerciales pero no en muchos editores. Lo más conveniente es considerar a los editores como caso particular de los procesadores de palabras; es decir, casi todos los procesadores de palabras tienen las capacidades de un editor además de las propias.

Funciones de un editor

Estas son algunas de las funciones de los editores actuales:

- Creación de un archivo que contendrá texto.
- Facilidades para manipular el archivo, que incluyen, entre otras:
  - — Agregar texto a un archivo mediante el teclado de la terminal.
  - — Capacidad de borrar caracteres o palabras, renglones completos y grupos de ellos.
  - — Capacidad de insertar caracteres, palabras o renglones en medio de texto ya existente.
  - — Capacidad de buscar en el texto algún subtexto en particular y, posiblemente, cambiarlo por algún otro.
  - — Capacidad de desplegar en pantalla cualquier porción del texto, determinada por número de renglón, número de página, contenido de un renglón, etc.
  - — Capacidad de copiar porciones de texto a otros lugares dentro de ese archivo o de algún otro, así como la acción inversa: traer texto de otros archivos para integrarlo al actual.
- Combinación de varios archivos en uno solo y su inversa.
- Creación de "ventanas" para tener acceso a contenidos de diferentes archivos en forma simultánea.
- Facilidades de edición, entre las que se incluyen paginación, notas al pie, índices, subtítulos y portadas, alineación del margen derecho y, en algunos casos, búsqueda (y corrección) de errores tipográficos y ortográficos.
- Eliminación de un archivo.

Existe una gran diversidad de editores, que van desde los más elementales (que no cubren todas las funciones recién expuestas) hasta verdaderos sistemas completos de edición, o de ayuda para creación de programas en algún lenguaje específico de programación, que detectan cierto tipo de errores léxicos y sintácticos al momento de la creación del programa. Igualmente, hay editores programables en los que el usuario especifica una tarea por realizar (cambiar todas las apariciones de una palabra en cierto contexto por alguna otra, por ejemplo) y el editor se encarga de aplicarla a todo un conjunto de

archivos; también hay editores guiados por un menú, del que el usuario escoge una opción, sin tener que manejar un lenguaje especial.

En vista de que un editor es un programa que interactúa directamente con el sistema operativo (porque se tiene que encargar de que las órdenes que el usuario da pasen al sistema de archivos de la computadora en la forma más eficiente posible), varios de ellos ofrecen acceso directo a las funciones del sistema operativo sin abandonar el editor, lo cual es muy conveniente en muchos tipos de aplicaciones complejas en las que el editor se considera una extensión del sistema operativo. Así, existen casos en los que se crea un programa mediante el editor, por ejemplo, en el lenguaje Pascal, que a continuación se compila. Si hay errores de sintaxis, el editor toma el control y muestra al usuario la línea del programa que contiene el error, junto con el diagnóstico apropiado. De esta forma se ligan las funciones de edición y compilación en un todo que parece más natural. Lo mismo sucede en algunos procesadores de palabras, que detectan errores tipográficos (o de construcción, como uso excesivo de una misma palabra o sílaba, etc.) e incluso sugieren la corrección apropiada que, entonces, se integra de inmediato al texto en proceso.

Todas estas capacidades del editor deben traducirse a operaciones internas de modo que la computadora pueda efectuarlas a la mayor velocidad posible. Está claro que esto implica que el texto en proceso debe residir en la memoria central de la computadora, el único lugar donde se garantiza la velocidad de proceso. Aquí, sin embargo, vuelven a surgir problemas como los que nos ocuparon anteriormente. La capacidad de la memoria central es relativamente pequeña y, por tanto, no se puede esperar que los textos de los múltipes usuarios que están empleando el editor de una computadora puedan residir completos en la memoria al mismo tiempo. Esto obliga a que en el diseño del programa editor se considere el texto como una cadena de renglones, ligada mediante lo que se conoce como una estructura de datos; para que el programa pueda trabajar sobre grupos de ellos, y sea capaz de traerlos y llevarlos en forma dinámica de y hacia el disco magnético sin detener la operación ni degradar demasiado el tiempo de respuesta.

Casi todos los editores, entonces, primero crean una copia del archivo que reside en el disco magnético —cuando se trata de un archivo que no es de nueva creación— y la colocan (o al menos parte de ella) en la memoria central de la computadora, y es sobre esta copia sobre la que se trabaja. Al final de la sesión de edición el archivo se lleva de regreso al disco, en donde reemplaza a la versión original. Algunos ofrecen la capacidad de "recrear" la sesión de edición completa para el caso de que haya habido alguna falla en el equipo o un corte en el suministro de corriente eléctrica, recuperando así el trabajo hecho por el usuario.

Casi todos los editores actuales son de un tipo conocido como **editores de pantalla**, en los que en la terminal de video aparece una página virtual que representa una porción del texto. Sobre esta página que aparece en la pantalla es posible realizar las operaciones ya descritas, mediante órdenes que el usuario da al editor empleando las teclas de la terminal, o combinaciones de ellas. Resulta muy conveniente emplear uno de estos sistemas, porque en efecto se trabaja con una imagen virtualizada del archivo, en la que es posi-

ble mover y transformar porciones a voluntad (usando la pantalla como ventana al archivo) y lograr así resultados que de otra forma requerirían mucho esfuerzo o serían imposibles.

Los primeros editores eran de un tipo conocido como **editores de línea**, en los que la ventana mostraba sólo uno o pocos renglones de texto y las modificaciones deseadas no se aplicaban directamente sobre ellos, sino que era forzoso describirlas antes mediante instrucciones especiales; esto hacía bastante complicada la labor de moverse a lo largo y ancho de las páginas que componen el texto. En vista de las ventajas que representa trabajar con un editor de pantalla, los de línea han caído en desuso.

En términos generales, el diseño de un editor representa un esfuerzo considerable de ingeniería de software, y son muchos los recursos técnicos y teóricos que deben intervenir en su elaboración, y ésta es una de las razones por las que la venta de este tipo de sistemas ha tenido gran auge, ya que la creación de un editor es un área especializada, bastante lejana de las labores usuales de un centro de procesamiento de datos.

Bases de datos

Un **sistema administrador de bases de datos** (DBMS por sus siglas en inglés, y que aquí abreviaremos como **SABD**) es un gran programa que tiene como función virtualizar el manejo de archivos de datos para permitir que el acceso a la información que contienen se lleve a cabo en forma lógica y no física. Es decir, mediante una base de datos (que es también un nombre abreviado de estos sistemas) es posible procesar archivos guiados por el significado de sus contenidos, y no tanto por sus características de organización (registros, campos, tamaños, etc.).

Diferencias entre datos e información

Para comprender la razón de ser de las bases de datos es necesario antes hacer una distinción básica entre **datos e información**. Por lo primero se entiende tan sólo la representación (normalmente en forma magnética) de números y letras o palabras, sin que su significado sea la característica más importante, mientras que por información se entiende esos mismos datos más las **relaciones estructurales** entre ellos. O sea, puede haber datos sin información (simples números), pero no puede haber información sin datos, pues la información se extrae de los datos mediante relaciones explícitas que se proponen. Un conjunto de números, por ejemplo, en tanto simples cifras, no significa mayor cosa, pero si se decide averiguar algunas relaciones estadísticas entre sus componentes (distribución, coeficientes de crecimiento, etc.), entonces la información aparece justamente allí donde antes había tan sólo datos, y puede incluso servir para determinar acciones a realizar, suponiendo que representen mediciones de algún parámetro. El conjunto de números (banco de datos) no ha cambiado; lo que sucedió fue que se hicieron explícitas algunas relaciones entre sus elementos. En principio, el conjunto y diversidad de relaciones que se puede establecer en un banco de datos es prácticamente ilimitado y depende del observador que las propone, y no tanto de los datos mismos.

Lo que se obtiene con el uso de una base de datos finalmente es una elevación del nivel de abstracción de los datos que se manejan, es decir, de la forma en que se observa la información. Los niveles más bajos de abstracción están muy ligados con la arquitectura física de la computadora y de los dispositivos de almacenamiento secundario. Esos niveles son bastante difíciles de manejar, y requieren que se conozca bien la computadora y el sistema operativo, lo que hace poco popular la programación de aplicaciones para manejar información en esta forma. Conforme se alcanza un mayor nivel de abstracción en el manejo, los usuarios se apegan a una realidad en particular de su interés, como pueden ser empleados, departamentos, compras, pólizas, etc.; y al interactuar con el sistema plantearán las operaciones en términos de esa realidad, sin tener que ocuparse de los detalles operativos.

El origen de los manejadores de bases de datos puede situarse entonces en la gran cantidad de información que se puede extraer de un conjunto de datos dado, lo cual vuelve muy deseable disponer de una especie de extractor general de información a partir de allí. Para lograr este objetivo se tendrá que contar con programas o sistemas que se encarguen de lo siguiente:

Funciones de una base de datos

- Creación del **banco de datos**, organizado en alguna forma (registros, campos, etc.), que quedará residente en algún medio accesible a la computadora. Esto se logra normalmente mediante formatos especiales que aparecen en la pantalla para captura de datos.
- Definición del **esquema** de organización, que describe la estructura de la información junto con los formatos, longitudes y tipos de agrupación de los datos que conforman el banco.
- Manipulación del esquema, en términos cercanos al problema que se representa, y no tanto en función de las características de organización de los datos.
- Obtención de informes y resultados a partir de los datos y las relaciones definidas entre ellos.

Entonces, el mecanismo general con el que se obtiene información mediante un sistema administrador de una base de datos es el siguiente:

Cómo se interactúa con una base de datos

1. El usuario plantea sus requerimientos de información y las fuentes para obtenerla, y propone la forma que deben tener los informes que desea.
2. El analista determina lo que se conoce como **estructura lógica** de la base de datos: un mapa en el que se hacen explícitas las relaciones entre los componentes de la información, y a partir del cual se podrán extraer resultados útiles para el usuario. Otro nombre que suele darse a esta representación es *esquema de la base de datos*. Esta es la labor más importante y la que requiere de un conocimiento formal sobre lo que en ciencias de la computación se conoce como diseño de bases de datos, que se menciona más adelante.
3. Ya en la computadora, el sistema administrador de bases de datos recibe una solicitud de operación escrita en un lenguaje especial de manipulación de datos y ésta es procesada, por un traductor que forma parte

del SABD, a formas más ligadas a las particularidades de la computadora o sistema operativo existente.
Puesto que los datos deben existir aun cuando no se estén ejecutando los programas, su estructura y su forma también deben estar siempre presentes para el sistema manejador de bases de datos y las partes que lo constituyen. Esto se logra especificando en un lenguaje especial (conocido como **lenguaje de definición de datos**) la organización física de los datos que constituirán la base de datos, para que entonces un traductor de este lenguaje la procese; el resultado es una representación interna de la estructura de la base de datos. Esta representación contiene, entre otras cosas, las relaciones que se hicieron explícitas entre los datos, expresadas mediante algún esquema o lenguaje (relaciones de dependencia o jerarquía, de inclusión, de ordenamiento, etc.) y los resultados que se desea obtener al manipular los datos.

4. El operador del sistema (o incluso el usuario mismo) hace pedidos específicos de información al SABD, que pueden involucrar una gran cantidad de operaciones internas, llamadas al sistema operativo, y operaciones de E/S y acceso a los sistemas de archivos. Para que los resultados de la operación sean accesibles, tienen que ser traducidos de nuevo al nivel de abstracción que le corresponde al usuario. La tendencia actual en los sistemas es que esos pedidos puedan ser expresados en un lenguaje relativamente sencillo (conocido como **lenguaje de consultas**, *query language*), que permite al usuario ser quien explote la base de datos desde sus propias terminales. A veces es necesario, cuando la aplicación es compleja, escribir en algún lenguaje de programación compatible con el SABD, programas para explotar características específicas del sistema, que no son fácilmente accesibles desde el lenguaje de consultas. Esta es una tarea especializada que normalmente queda fuera del alcance del usuario final.

Diseño de bases de datos

Una base de datos* debe reflejar la información de una realidad que es importante para una organización. El diseño de la forma y estructura de la base de datos es un proceso muy importante, ya que de esto dependerá la simplicidad o complejidad de los programas de aplicación, y lo compacta o redundante que sea la información dentro del sistema.

Para poder hacer un diseño adecuado es necesario simular de alguna forma una situación del mundo real en la computadora. La simulación se lleva a cabo a través de un modelo de datos. Un **modelo de datos** es una herramienta conceptual que ofrece maneras de especificar las entidades de información y las relaciones que existen entre ellas. Quienes se encargan de dise-

* Puede existir confusión en el uso de este término: a veces *base de datos* se refiere a un SABD, y a veces se refiere tan sólo al diseño de los archivos de información sobre los que trabaja. Normalmente el contexto permite determinar el uso correcto. En este caso, por ejemplo, se trata del diseño de los archivos y su interrelación lógica.

ñar la base de datos son los analistas de sistemas a cargo de un proyecto en particular.

Puesto que los sistemas administradores de bases de datos manejan información y los operadores para manipularla, están íntimamente relacionados con algún modelo de datos en particular. Existen varios modelos de datos, que permiten mayor o menor capacidad y facilidad para el diseño de alguna base de datos específica. Entre los modelos de datos destacan el **modelo de entidades y asociaciones** (que es el más general, aunque hay pocos SABD que lo emplean), el **modelo de redes**, el **modelo jerárquico** y el **modelo relacional**. Este último es el más empleado en la actualidad y el que más futuro tiene en términos prácticos, puesto que está formulado con bases teóricas y formales más sólidas y mejor especificadas, lo que permite la existencia de SABD relacionales poderosos y concisos, que funcionan incluso en microcomputadoras.

El modelo de datos determina las características generales del SABD

Las bases de datos constituyen, como se ha dicho, todo un campo de estudios dentro de las ciencias de la computación, por lo que existen cursos dedicados por completo a este tema, así como estudios de posgrado y especialización. Existe también un gran número de textos dedicados a los aspectos formales y operativos de las bases de datos, entre los que se puede citar [DATC86], [ULLJ82], [WIEG83], todos ellos de gran amplitud y grado de especialización.

Hojas de cálculo

Se conoce como **hoja electrónica de cálculo** (*spreadsheet*) a unos sistemas que permiten el manejo virtualizado de columnas de números, y que vuelven fácil la tarea de hacerles modificaciones y operaciones diversas, que van desde alteraciones sencillas en sus valores hasta el cálculo de cifras adicionales que dependen de relaciones matemáticas entre otras columnas y renglones, especificados por el usuario.

La utilidad de mantener columnas de números (sin considerarlas como pertenecientes a una matriz, sino desde otra perspectiva menos formal) es muy amplia, sobre todo en aplicaciones de contabilidad, finanzas y presupuestos. Las primeras hojas de cálculo para computadoras personales tuvieron tanto éxito que muy pronto se crearon compañías especializadas en ese campo. Actualmente existen hojas de cálculo muy complejas y vistosas, sobre todo para microcomputadoras, pues manejan interfaces con procesadores de palabras, bases de datos y sistemas de graficación a color.

La hoja de cálculo es una forma bidimensional que consiste en celdas almacenadas en la intersección de renglones y columnas, en donde se puede guardar, borrar o reacomodar información, que luego se puede manipular en diversas maneras. El método usual que se emplea para esa manipulación es mediante la definición de **fórmulas aritméticas** que relacionan columnas o renglones. Así, es factible definir, entre muchas otras cosas, fórmulas aritméticas que usen algunas de esas celdas como variables de una ecuación que calcula el retorno esperado de una inversión dentro de un plan financiero,

o que lleva el control de las existencias dentro de un almacén. También es sencillo formular informes a partir de esas cifras porque todos estos sistemas tienen facilidades para poner títulos a las columnas y para imprimirlas en algún formato especificado por quien maneja la información.

Por ejemplo, si se tiene una columna de números que representa los sueldos de los empleados de un departamento, una sencilla fórmula que indique que cada elemento se multiplica por 1.30, y otra que indique que se suma toda la columna, proporcionará un elemental estudio de un incremento salarial del 30%, tanto en su impacto individual como global.

Funciones de una hoja de cálculo

Entre las funciones usuales que una hoja de cálculo permite efectuar se encuentran las siguientes:

- Ingreso de números o textos en celdas
- Modificación de valores de celdas
- Copia de celdas de un lugar a otro dentro de la hoja
- Cálculo de valores de acuerdo con fórmulas especificadas
- Modificación de valores en secciones completas de la hoja
- Especificación de funciones diversas que afectan a secciones de la hoja, como determinación de promedios, cálculo de valores máximos o mínimos, detección de celdas con valores preespecificados, y otros
- Copia e integración de varias hojas en una sola
- División de una hoja
- Impresión de la hoja con un formato determinado

Desde un punto de vista computacional, una hoja electrónica de cálculo no es más que un gran arreglo de celdas (donde cada una contiene una cifra) que están ligadas entre sí mediante estructuras de datos internas que permiten su manipulación interactiva desde la pantalla, para que el usuario pueda relacionarlas de múltiples maneras.

A diferencia de las bases de datos, las hojas de cálculo normalmente sí son definidas y operadas por los directamente interesados en la información que contienen, y esto es posible gracias a que su diseño y presentación se pensó para que fueran usadas por los propietarios de las computadoras personales. El lector no debe dejar que esta facilidad de uso le impida enterarse de que para diseñar una hoja de cálculo se requiere una gran cantidad de conocimientos de programación, estructuras de datos e ingeniería de software, y esto ya no debe parecernos raro; la programación de sistemas es una rama muy sólida de las ciencias de la computación que tiene como finalidad escribir sistemas que den al usuario una imagen de transparencia y sencillez tal que escondan la complejidad de su diseño.

4.8 Inteligencia artificial

En este largo camino de esfuerzos de comunicación entre la máquina y el ser humano se puede vislumbrar una última etapa, que debería ser considerada como la finalidad de la programación de sistemas: la comunicación directa

con la máquina, en lenguaje natural, y sin el complicado conjunto de lenguajes intermedios que se han descrito a lo largo del capítulo. Esta idea es, en principio, inalcanzable en toda su extensión, ya que implicaría la capacidad de reproducir a la perfección los extraordinariamente complejos mecanismos lingüísticos (entre otros) que nos caracterizan, lo cual requeriría una virtual reinvención del ser humano, tarea que evidentemente no nos es accesible. Sí es posible, sin embargo, acercar aun más la computadora a nosotros, simulando —aunque sea en grado imperfecto y limitado— algunas de las aptitudes verbales, de razonamiento y de entendimiento que poseemos; y ésta es precisamente la tarea del campo de técnicas y conocimientos que se identifica como **inteligencia artificial**.

Lo que no es la inteligencia artificial

Debe quedar claro que inteligencia artificial no implica computadoras inteligentes; implica más bien computadoras que ejecutan programas diseñados para simular algunas de las reglas mentales mediante las cuales se puede obtener conocimiento a partir de hechos específicos que ocurren, o de entender frases del lenguaje hablado, o de aplicar estrategias para ganar juegos de mesa. En ningún caso se habla (o siquiera se vislumbra) de la capacidad de entender realmente situaciones elaboradas o complejas, o mucho menos aun, de siquiera acercarse a tener independencia o remotamente a tener sentimientos. La complejidad inherente a estas últimas funciones es tan enorme que no las entendemos siquiera desde un enfoque psicológico, y mucho menos desde un punto de vista fisiológico.

Esto no impide que la inteligencia artificial avance con gran rapidez, y ya se tienen básicamente resueltos los problemas de entendimiento de frases cortas habladas (mas no de historias o razonamientos complejos), del camino a seguir en juegos de mesa que requieren análisis de estrategias y de la capacidad de decir algo coherente (o incluso hacer sugerencias) acerca de situaciones específicas que han sido descritas a la computadora. Ha resultado mucho más difícil —y de hecho allí los avances han sido menores— el campo de la traducción automática de idiomas, el reconocimiento de formas complejas tridimensionales, la creación de robots con capacidades generales de movimiento, y el entendimiento de relatos con argumentos no triviales, como se describe en el artículo [WINT84].

Tal vez el progreso más visible se ha dado en lo que se conoce como **sistemas expertos**, que es el nombre genérico para los programas especializados en algún campo específico del conocimiento, y que tienen la capacidad de simular razonamientos parecidos a los que haría una persona versada sobre el tema en cuestión. En actividades como la explotación de pozos petroleros o la autorización de cuentas en tarjetas de crédito, entre otras, se pide a estos sistemas (mediante lenguajes especiales de comunicación a través de la terminal de video) que propongan caminos a seguir para la toma de decisiones. Un sistema experto puede dictaminar la conveniencia de iniciar una exploración preliminar en busca de petróleo a cierta profundidad, una vez que ha analizado los datos geológicos adecuados, por ejemplo, o bien puede sugerir que no se autorice una transacción de una tarjeta de crédito debido a que la historia reciente de ese cliente es errática y no muestra un patrón de compras previsible. Es importante aclarar que un sistema experto opera

sobre un solo campo de conocimientos y que, por tanto, está dedicado exclusivamente a eso.

Estructura de un sistema experto

La estructura interna de un sistema experto está basada en lo que se conoce como "máquina de inferencias": un programa capaz de manejar el conjunto de reglas de razonamiento necesarias para llegar a una conclusión que no existe en ese momento sino que se generará como resultado del análisis. El mecanismo de inferencias puede reconstruir hasta cierto grado los pasos lógicos que hay que dar para extraer conocimiento de un conjunto de datos y de las reglas predefinidas para su manipulación. Para lograr lo anterior el sistema contiene lo que se conoce como una **base de conocimientos** (término derivado de *base de datos*) formada por reglas y atributos, y un procedimiento para su evaluación y aplicación, ligado con un subsistema gramatical para la presentación de preguntas y resultados por la pantalla. (Para una descripción técnica de lo que constituye un sistema experto, véase el artículo [BEVT85].)

Existen sistemas que pueden aprender con base en experiencias recién adquiridas, por ejemplo, en juegos de estrategia; y en inteligencia artificial se habla también de "programación heurística", que se refiere a técnicas de búsqueda y análisis que producen comportamientos menos rígidos que los que de ordinario se esperan de un programa de computadora, porque internamente determinan el camino óptimo a seguir para llegar a un fin propuesto.

En el caso de sistemas matemáticos sobresalen los que pueden manipular y demostrar algunos teoremas y los que tienen capacidades algebraicas, casi ilimitadas, que les permiten hacer derivación e integración simbólica, resolver series y simplificar y transformar ecuaciones extraordinamente complejas. En el artículo [PAVR81] se muestra cómo uno de estos sistemas encontró tres errores en un gigantesco conjunto de ecuaciones empleado el siglo pasado por el astrónomo francés Charles Delaunay para determinar la órbita lunar; la cantidad y complejidad de ese sistema de ecuaciones de mecánica newtoniana era tal que el autor dedicó 10 años a definirlas y otro tanto para verificar sus resultados. Un sistema computacional algebraico analizó ese trabajo de 20 años (contenido en 2 volúmenes) y luego de 20 horas encontró dos errores menores y uno sustancial, ninguno de ellos numérico, todos simbólicos.

Este tipo de sistemas, y en general casi todos los relacionados con la inteligencia artificial, suelen estar escritos en lenguajes de programación especializados (que se mencionan en el capítulo 7 de este libro), entre los que destacan LISP y Prolog, aunque en principio cualquier lenguaje de programación podría servir para este propósito.

Como ejemplo de lo logrado hasta ahora en el área de simulación de algunos procesos mentales, considérese que una computadora ganó el campeonato mundial del juego de salón de origen árabe conocido como *backgammon*, celebrado en Montecarlo en 1979 [BERH80]. Las máquinas que juegan ajedrez lo hacen en un nivel de aproximadamente 1900 puntos [DOUJ78], lo cual las coloca en la categoría de expertos (que, aunque está por encima de la media nacional de países como Estados Unidos, sigue aún muy por debajo

de los niveles de competencia humana internacional; cuando fue campeón mundial, "Bobby" Fischer tenía una clasificación superior a 2700). Los grandes bancos emplean ya como rutina sistemas expertos para el análisis de cuentas de inversión, y lo mismo sucede con empresas petroleras, con los diagnósticos en algunos hospitales especializados y con la determinación de riesgos y pérdidas en líneas de producción automatizada de ensambles para la industria aeronáutica. En el artículo [DEWA84] se describen algunas de las estructuras de funcionamiento de programas que juegan damas y que están ya en un nivel de competencia de campeonato mundial.

En el enfoque que hemos propuesto, la inteligencia artificial ocupa el sitio más avanzado dentro de la programación de sistemas y, por ello, no debe resultar raro que sea un campo avanzado de conocimientos dentro de las ciencias de la computación. En general, la inteligencia artificial se estudia en cursos de posgrado, o en los últimos niveles de una carrera universitaria en computación, y se puede considerar como un campo abierto a las labores de investigación. La referencia [WINP84] constituye un ejemplo del nivel de especialización de este tipo de estudios; por otro lado, la revista *Byte* dedicó el número de abril de 1985 al tema de la inteligencia artificial y constituye una referencia mucho más accesible, en la que se tratan temas sobre lenguajes de programación dedicados, máquinas especiales, visión, aprendizaje y sistemas expertos. Incluye además un artículo de Marvin Minsky, autor pionero de este campo que también será mencionado por sus trabajos sobre computabilidad en el siguiente capítulo.

4.9 Resumen del capítulo

Si el lector siguió con detenimiento las explicaciones que se han dado a lo largo de este capítulo, al menos tendrá claras dos cosas: una computadora no es tal sin todo el complejo de programas del sistema que la acompañan y, la programación de sistemas es un área especializada dentro de las ciencias de la computación, que puede alcanzar niveles de profundidad que rebasan ampliamente los límites de nuestro libro y que se extiende incluso a niveles de estudios de posgrado.

Sin embargo, podemos intentar resumir aun más la información, porque en realidad la programación de sistemas es un tema que concierne sobre todo a los diseñadores y estudiosos de las ciencias de la computación, y no tanto a los usuarios de una computadora. Como se expresó claramente en la introducción de esta obra, cuanto más sepa un usuario acerca de la filosofía y modo de operación de un sistema de cómputo, mejor uso potencial podrá hacer tanto de él como de las posibilidades de desarrollo que le ofrece.

Hemos dicho que una computadora tiene capacidades bastante limitadas en lo que se refiere a la cantidad y diversidad de operaciones elementales que puede procesar, pero que es capaz de ejecutarlas a enormes velocidades. Esto da lugar a una aparente paradoja: se dispone de una máquina que rebasa inimaginablemente al ser humano en velocidad y capacidad de proceso, pero

sus habilidades son tan restringidas que no resulta de mucha utilidad práctica.

¿Qué hacer para aprovecharla cabalmente? La respuesta está en la programación de sistemas, que utiliza precisamente esas virtudes electrónicas de velocidad para enriquecer en forma gradual la complejidad de sus operaciones. La idea detrás de esto es sencilla, y consiste en armar herramientas de programación que se integren a la computadora y la hagan parecer dotada de capacidades cualitativamente superiores.

El papel de la programación de sistemas

En la sección 4.1 se explicó cómo es y qué se puede esperar del lenguaje nativo —por así decirlo— de las computadoras, llamado lenguaje de máquina. Se vio que, por sus características constructivas, la unidad central de procesamiento puede ejecutar un pequeño número de instrucciones de máquina, que le son particulares, y cuyo radio de acción alcanza tan sólo a los elementos que la constituyen (registros, acumulador, celdas de memoria central, etc.). Las desventajas, pues, de hacer programas en lenguaje de máquina son múltiples, y pueden resumirse así:

Lenguaje de máquina

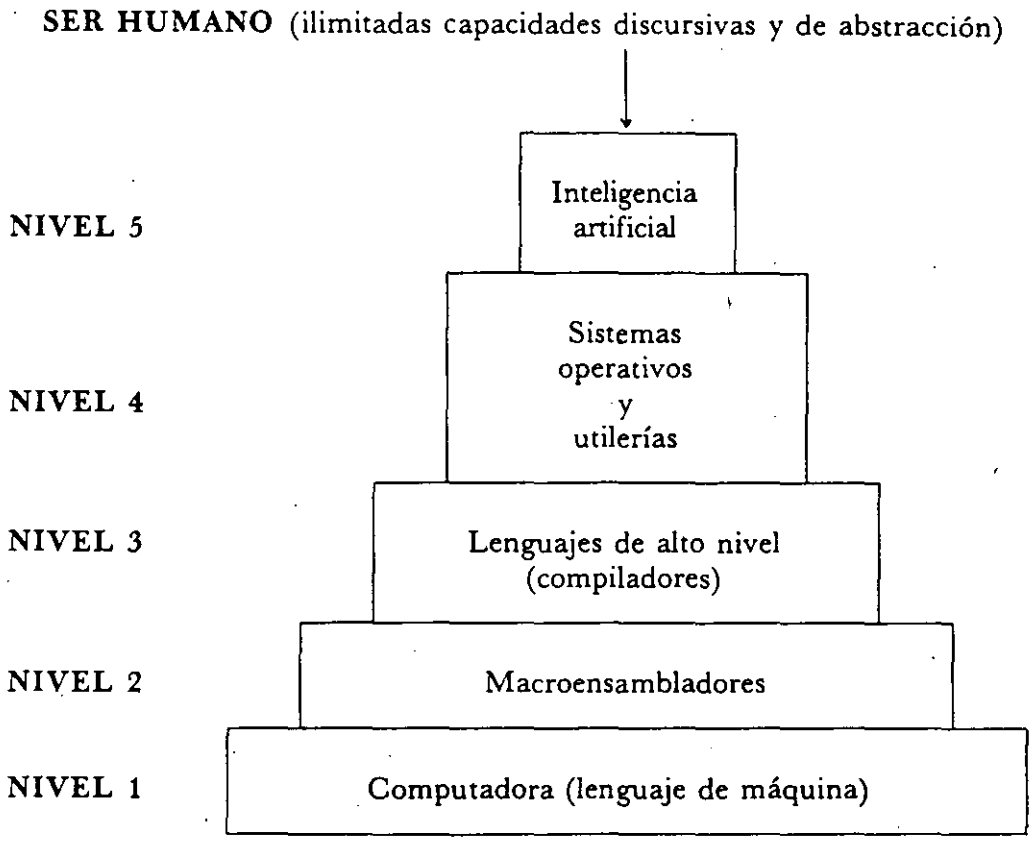
- Un programa en el lenguaje de una máquina no puede ser "entendido" por otra de características diferentes.
- Un programa en lenguaje de máquina resulta por completo incomprensible para un ser humano ya que, por definición, forma parte de los llamados programas objeto, que son los únicos ejecutables directamente por un procesador, y que están escritos en sistema binario.
- La programación de este tipo no puede alcanzar niveles considerables de complejidad, ya que carece casi por completo de estructura y contenido semántico, puesto que toda instrucción de máquina no puede sino remedar las particularidades y limitaciones de la unidad central de procesamiento.

La única ventaja de hacer programas en lenguaje de máquina es, por supuesto, que se logra que una computadora efectúe las tareas que se le encomendaron, y que lo haga de forma automática y a gran velocidad.

El desafío está claro: hay que dar respuesta al problema con que nos enfrentamos, y que se puede incluso caracterizar en términos filosóficos si nos detenemos a considerar la razón de ser de las computadoras. Proponemos la idea (que será desarrollada en el siguiente capítulo) de que una máquina de esta clase tiene posibilidades de servir no sólo como instrumento de cálculo, sino como herramienta que puede extender las capacidades de la mente y ayudarla a modelar en forma adecuada la realidad, con fines de mejorarla.

Pues bien, como parece que existe una diferencia considerable entre las computadoras y nosotros, tanto en velocidad como en capacidad y alcances, requerimos de un hilo conductor que sirva de guía en el estudio del problema de la comunicación entre hombre y máquina, y éste no puede ser otro que lo que llamaremos la *distancia gnoseológica* (en términos de teoría del conocimiento) entre ambos. Por esto nos referimos a las capacidades de abstracción, por un lado, y a la facilidad de comunicación, por el otro. Está claro que, a medida que se utilizan menos recursos para entablar comunicación con una

computadora (como en el caso del lenguaje de máquina), menor resulta la calidad de lo que se puede comunicar; y si se requiere un alto nivel de comunicación que incluso maneje ciertas formas de abstracción, hay que pagar el precio derivado precisamente de esa distancia. Esto resulta muy claro al hacer el análisis de la función de los ensambladores, compiladores y sistemas operativos, de acuerdo con el siguiente esquema, que ilustra simbólicamente los trechos que hay que abarcar para que, partiendo de los niveles de comunicación a los que estamos acostumbrados, se llegue a establecer contacto con una computadora.



Se han resumido ya las limitaciones del lenguaje de máquina y, partiendo de ellas, se analizará del mismo modo el problema de los ensambladores.

La idea de un ensamblador consiste sencillamente, como se dijo en el apartado 4.2, en que la propia computadora sea la que traduzca las expresiones escritas en lenguaje ensamblador, a lenguaje de máquina. El lenguaje ensamblador dispone de algunas facilidades adicionales sobre el limitado lenguaje de máquina, pues permite trabajar con cierta independencia de la arquitectura (configuración física) de la unidad central de procesamiento, aunque tampoco constituye, de ninguna manera, la respuesta al problema de la distancia que nos separa de un sistema de cómputo en este segundo nivel.

Las desventajas principales de la programación en lenguaje ensamblador, desde el punto de vista del ser humano, son:

Lenguaje ensamblador

- Un programa escrito en el lenguaje ensamblador de una máquina no puede ser "entendido" por otra de tipo diferente.
- El lenguaje ensamblador sigue dependiendo en gran medida de las particularidades de la unidad central de procesamiento y las celdas de memoria, lo que vuelve difícil y penosa la tarea de hacer programas complejos.
- Es prácticamente imposible mantener la estructura y riqueza expresiva de una idea cuando ésta se expresa en lenguaje ensamblador.

Por otro lado, sus ventajas respecto al primer nivel (lenguaje de máquina) son evidentes: libera al programador de la dependencia total de las direcciones absolutas de memoria y le permite la posibilidad de emplear variables simbólicas y etiquetas en sus programas.

Se dijo además que es posible enriquecer sustancialmente la idea del ensamblador si se le añade la capacidad de manipular grupos de instrucciones como si fueran una sola unidad, dando origen al concepto de los macroprocesadores y los macroensambladores. Por medio de esta nueva idea ya somos capaces de comunicarnos con la máquina sin necesidad de repetir en los programas conjuntos de instrucciones que son necesarias, dejando esta tarea al nuevo traductor.

Como resulta comprensible, el método "genético"* que se está empleando no es sin costo: aunque resulta más atractivo para nosotros trabajar en este segundo nivel, no lo es tanto para la máquina, que tiene la tarea de traducir primero a lenguaje de máquina todo lo que se le dice, para poder entonces cargarlo a la memoria y ejecutarlo. Es por esto que se dedicó un espacio (Sec. 4.4) a describir la idea fundamental de los cargadores, cuya función consiste precisamente en tomar un conjunto de instrucciones de máquina almacenadas, por ejemplo, en el disco magnético, y depositarlas en la memoria central.

Si se desea dar otro paso hacia arriba habrá que detenerse y considerar con mucho cuidado las tareas que siguen, porque implican la capacidad —por primera vez— de expresar ideas en términos dotados de una estructura, como se explicó en el apartado 4.5. Esto significa que hay que "enseñar" a la computadora a analizar frases completas, y ya no simples conjuntos de instrucciones del procesador, lo cual se logra por medio de un traductor especializado llamado compilador. Los problemas teóricos a los que hubo que enfrentarse son de magnitud tal que todavía siguen despertando interés entre la comunidad académica internacional, pues de ninguna manera está cerrado el desarrollo en este campo.

Como ya se señaló, existen múltiples lenguajes de programación de alto nivel expresivo, que comparten, de alguna u otra manera, las siguientes ventajas con respecto al nivel anterior:

* Lo llamamos así para enfatizar su característica evolutiva; es decir, un nivel superior está formado con los elementos que el anterior hace posibles, y los emplea de forma integral para ser lo que es. El concepto mismo de herramienta tiene estas características, ya que por medio de las más primitivas es posible construir otras más avanzadas, que a su vez servirán para continuar el proceso.

Lenguaje de alto nivel

- En principio, es posible compartir un programa escrito en alguno de estos lenguajes con cualquier computadora que disponga del compilador adecuado (que convierte el programa fuente original en un programa objeto directamente ejecutable por el procesador, o en un programa equivalente escrito en ensamblador, que luego se traduce)
- Ya es posible respetar, en buena medida, la estructura original de una idea descrita detalladamente, y más aun si ha sido estructurada por medio de una metodología adecuada.

Sin embargo, sigue siendo necesario un entrenamiento formal para adquirir la capacidad de programar (que es, además, la razón de ser de este libro), y las computadoras aún están lejos de los niveles de abstracción y comunicación que usamos normalmente.

Si se mira hacia atrás se podrá distinguir que en este tercer nivel las cosas han cambiado bastante, pues ahora todo nos resulta más fácil, pero es más complejo para la computadora. Se requiere coordinar múltiples acciones, todas tendientes a que la máquina misma traduzca lo que se le expresa y paulatinamente lo baje de nivel, hasta llegar al lenguaje de máquina, que es lo único que puede procesar.

Esto ya es más que suficiente para decidirnos a estudiar alguna forma de reducir la complejidad de todas las tareas adicionales, lo cual lleva directamente al tema de los sistemas operativos.

El sistema operativo

La función general de un sistema operativo es controlar y dirigir la operación de las computadoras, de forma tal que presenten una imagen monolítica y virtual (en contraposición con real o electrónica o ingenieril) ante los usuarios —no sólo uno— del sistema de cómputo. Hemos mencionado que consideramos tan importante al sistema operativo como a las facilidades físicas mismas con que cuenta el equipo y que, de hecho, no se puede hablar de una computadora sin suponer implícitamente al sistema operativo que la controla y coordina.

Lo que se espera de un sistema operativo, *grosso modo*, es que sea perfectamente capaz de atender la operación concurrente de múltiples pedidos de atención por parte de procesos que estén en ejecución en la computadora; que sea perfectamente capaz de mantener toda la operación bajo control sin perder detalle alguno; que logre un óptimo grado de utilización de los recursos físicos de la máquina (procesador, memoria, periféricos) ... y, por último, que haga todo esto callada y eficientemente.

Este cuarto nivel de comunicación con la máquina no es, de ninguna manera, tan conveniente para los humanos como para los equipos, porque aquí hay que dedicar muchos recursos auxiliares para lograr la ejecución de la tarea original, cosa que es absolutamente justificable, dados los beneficios que reporta.

Inteligencia artificial

El último nivel del diagrama muestra un área que aún no adquiere importancia capital, pero que está destinada a desempeñar un papel relevante dentro de algunos años: la inteligencia artificial. Con este término no debemos aceptar implicaciones de ciencia ficción, sino un estudio científico y formal de algunos de los mecanismos con los cuales funcionan las capacidades hu-

manas de entendimiento, razonamiento y aprendizaje. Existen suficientes razones teóricas para denegar la idea de que las computadoras tomarán el control en algún momento, así como también existen razones para suponer que no está demasiado lejano el día en que se puedan encargar de una fracción de las tareas que actualmente recaen sobre nosotros, que van desde la automatización ("robotización") de muchos procesos de producción hasta la exploración del espacio exterior y el mejoramiento de las cosechas.

Por lo pronto, las tareas de la inteligencia artificial se han enfocado a la integración de sistemas dotados de capacidades limitadas de síntesis de voz, movimiento y percepción, así como al desarrollo de estrategias y esquemas de manejo de información (sistemas expertos) que incluyen la posibilidad tanto de externar opiniones autorizadas sobre temas específicos (perforaciones de pozos petroleros, geotermia, diagnósticos clínicos, etc.) como de aprender más sobre el tema.

Es mucho lo que se debe esperar en el futuro próximo, y de hecho existen esfuerzos de envergadura nacional para este propósito tanto en los Estados Unidos como en Europa y Japón, porque la siguiente generación de computadoras estará caracterizada por la inteligencia artificial. Para obtener un panorama amplio de lo que se puede esperar en los próximos años, tanto en el campo de la inteligencia artificial como en el de la tecnología de computadoras y sus aplicaciones en general, consúltese el número completo que la revista *Scientific American* dedicó a "La siguiente revolución de las computadoras", en octubre de 1987.

4.10 Anexo: cómo se diseña una computadora

Cuando se habla del diseño de una computadora puede pensarse en varios aspectos, en diferentes niveles operativos. Los más importantes son los siguientes, desde el más elemental hasta la integración final del equipo:

1. Diseño de los circuitos básicos (memoria, procesador, etc.)
2. Diseño de la arquitectura del sistema de cómputo
3. Diseño del software primitivo del sistema
4. Diseño o adaptación de software básico
5. Adaptación de software adicional

En virtud del enorme avance que ha registrado la microelectrónica, todas las computadoras actuales utilizan circuitos integrados en gran escala (VLSI) para resolver buena parte de las funciones básicas. En las mini y microcomputadoras, lo usual no es diseñar el procesador central (UCP) sino que se emplea algún microprocesador ya existente, así que la creación de una de estas computadoras parte del segundo punto de los recién mencionados. Esta tarea consiste en armar lo que se conoce como la arquitectura del sistema, y se logra mediante el acomodo e interconexión de microprocesadores, controladores, memorias y otros microcircuitos electrónicos, así como del diseño de las

interfaces requeridas. Esta es una labor altamente creativa y especializada, que podría considerarse similar a la construcción de un gran edificio empleando algunos materiales prefabricados. En este caso lo que podría llamarse la materia prima está representada por los circuitos VLSI, además de otros circuitos integrados en menor escala (de las familias SSI y MSI), que cumplen funciones de interconexión entre los circuitos principales; pero, claro, no basta con estos elementos, sino que tiene que haber inicialmente un diseño que los utilice para un fin preespecificado.

El tercer punto, software primitivo, es parte indisoluble de la creación de un equipo de cómputo, y se refiere a los programas en lenguaje de máquina que hacen que los microprocesadores y los controladores programables desempeñen sus funciones dentro de la arquitectura completa. En las computadoras actuales hay múltiples funciones de control que no son resueltas por el procesador central sino por microprocesadores dedicados que forman parte de la arquitectura completa, y es por ello que se vuelve necesario escribir estos programas internos de control. Por ejemplo, el manejo interno de entrada/salida en muchas computadoras se hace por medio de procesadores especiales que se encargan de leer y escribir bytes a una gran velocidad, sin que el procesador central se ocupe de ello. Lo mismo sucede con la transferencia de información a los discos rígidos y con algunas otras partes del sistema completo.

El cuarto punto se refiere al software de sistemas básico: cargador, ensamblador, compiladores y sistema operativo. Los sistemas básicos a veces se diseñan ex profeso para una nueva máquina (sobre todo en el caso de las máquinas grandes), o pueden ser adaptaciones muy especiales de sistemas ya existentes. Esta tarea ya no tiene mucho que ver con el diseño de la arquitectura, sino con su funcionalidad y los métodos básicos de operación, para aprovechar sus características específicas.

El último punto trata del conjunto de programas que le dan valor agregado a la computadora, como pueden ser procesadores de textos, hojas de cálculo, paquetería y otros. Muchas veces se encuentran ya en el mercado de software y, entonces, la tarea consiste en adaptarlos a la nueva máquina. Muchas compañías se dedican a esto y ofrecen adaptaciones de paquetes populares para muy diversas marcas de computadoras.

Por lo común, la palabra *computadora* se asocia con una máquina para procesar información, con terminales e impresora, pero esto no es necesariamente así. Existen computadoras de propósito especial para múltiples funciones, entre las que sobresalen comunicaciones y control de procesos, que no se parecen a las que suelen usar los programadores o usuarios, pero que son computadoras en todo el sentido de la palabra, porque tienen los mismos elementos funcionales que se han estudiado en el texto.

A continuación se describirá esquemáticamente el proceso de creación de una computadora comercial especializada para comunicaciones, que sirve como controlador para una red. Esta computadora se diseñó en un laboratorio local y ya está en producción industrial, también localmente. El objetivo de realizar esta descripción es mostrar al lector que esta tarea es factible y está al alcance de nuestras posibilidades.

El diseño de esta computadora para control de redes inicia a partir del paso 2 antes expuesto, e incluye también la creación del software primitivo, el tercer paso. En este anexo se presentará someramente el proceso de creación en sus diferentes etapas, mostrando los mecanismos de actuación y las motivaciones de los diseñadores, más que los aspectos técnicos.

Fases de creación

Diseño inicial

Naturalmente, la fase inicial de un proyecto, cualquiera que éste sea, consiste en definirlo con la mayor precisión posible; en crearlo mentalmente y definir sus características deseables o pedidas y, por ello, se comenzará por dar una idea de las motivaciones que dieron lugar a la concepción de esta computadora para control de redes y proceso distribuido.

Aun cuando el problema del proceso distribuido sigue recayendo en buena parte en los aspectos lógicos (de software), se debe contar con los medios físicos que permitan la interconexión de punto a punto necesaria para poder ejecutar adecuadamente tanto los procesos de control y administración de una red y sus recursos, como los múltiples programas de aplicación de los usuarios; así, se pensó en la necesidad de diseñar un dispositivo inteligente* que actúe como microcontrolador de comunicaciones, al que se dio el nombre de "anémona", porque tiene múltiples brazos y ramificaciones.

La anémona actúa como interfaz inteligente y cuenta con todos los requerimientos necesarios para el diseño y operación de una red local. Fue creada para cumplir con las características y flexibilidad necesarias para su adaptación en enlaces virtuales punto a punto dentro de diversos ambientes en que los elementos de la red pueden ser compatibles o incompatibles entre sí.

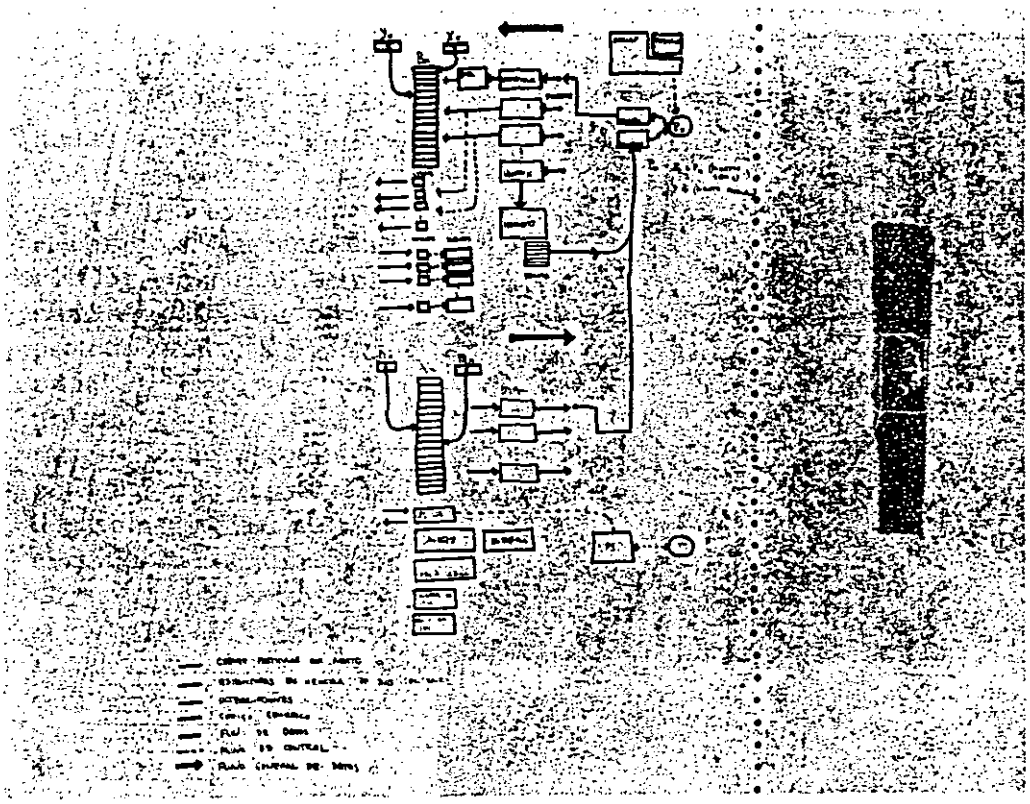
La idea fue obtener un dispositivo con 4 puertos serie sincrónicos o asincrónicos y dos paralelos, todos programables (es decir, que sus velocidades de transferencia sean definidas por el usuario), expandible en forma modular, y que pueda conectarse a un cable coaxial de bajo costo que hará las funciones de canal de comunicaciones.

Una vez que los diseñadores se ponen de acuerdo en la concepción del producto (y que por otra parte se ha determinado su viabilidad financiera) comienza el proceso de creación: qué tecnología emplear, qué arquitectura general proponer, qué características especiales habrá que definir, cuál será la forma de utilizarlo, etcétera.

Se decidió dotar a la anémona de un sistema interactivo de operación en línea que permita al usuario una utilización sencilla en el ambiente diario de trabajo sin perturbar las actividades de otros miembros de la red. Se decidió también emplear tecnología de bajo costo y circuitos integrados relativamente sencillos, para obtener un producto fácil de fabricar y mantener.

* A los dispositivos electrónicos controlados por medio de un microprocesador suele dárseles el nombre de "inteligentes", aunque esto no tiene nada que ver con la inteligencia artificial.

Durante esta etapa del proyecto el trabajo de los diseñadores consiste en proponer esquemas generales e ideas globales, que normalmente se pueden expresar en diagramas de bloques, y que sufren una buena cantidad de modificaciones y alteraciones en el transcurso de las discusiones que se forman a su alrededor. A continuación se muestran algunos ejemplos reales de este proceso.

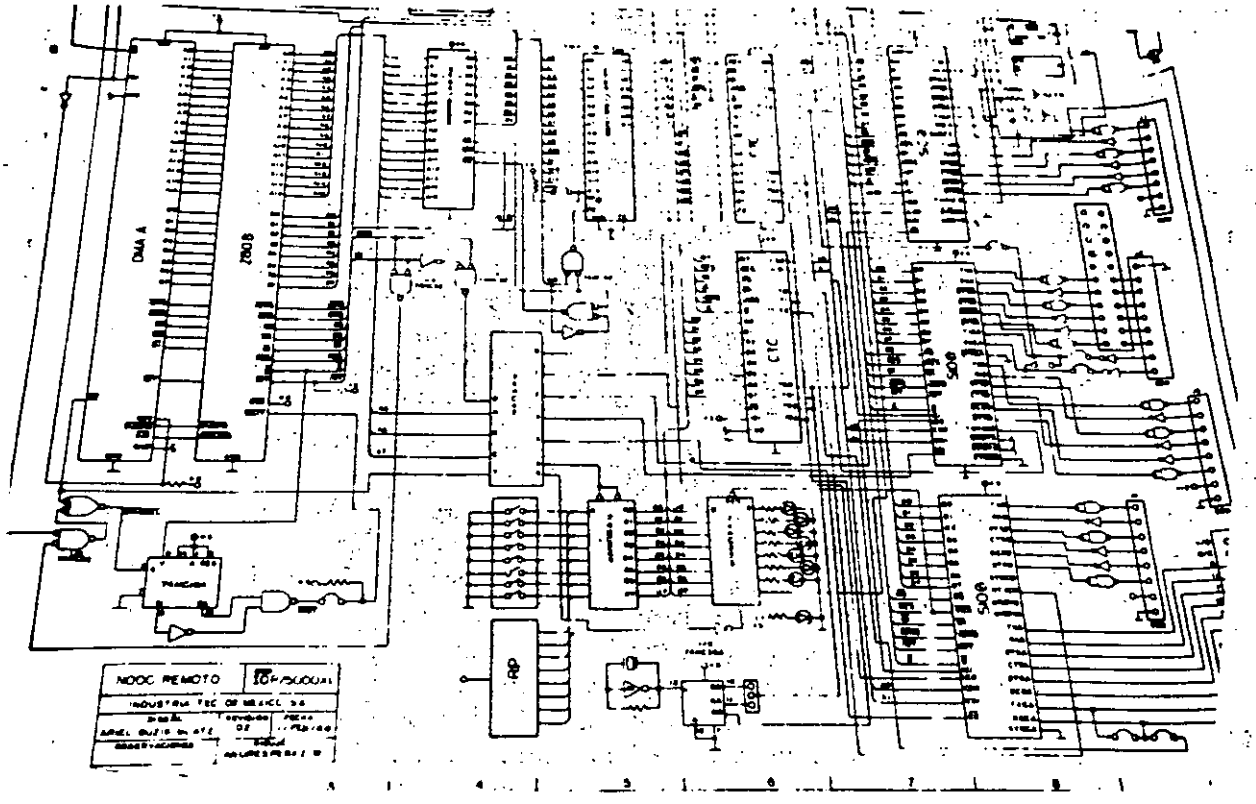


Diseño electrónico detallado y creación del software primitivo

Una vez que se ha aprobado el esquema global de la arquitectura, comienza la tarea de definir con mayor precisión sus componentes. Se reparte el trabajo entre varios equipos de ingenieros, que se dedican a definir cada uno de los módulos en términos de circuitos integrados, interfaces y componentes electrónicos. El diseñador principal debe mantener una vigilancia constante sobre cada equipo de trabajo, para asegurar —mediante reuniones periódicas— que se están siguiendo las especificaciones originales y que los diferentes subsistemas tendrán las mismas características eléctricas, lo que les permitirá interconectarse adecuadamente y transferirse datos e información. Quizá sea necesario revisar una o varias veces el esquema inicial para corregir errores o refinar algún detalle que no se consideró y esto surgirá durante las juntas de trabajo de todo el equipo.

Al final de esta etapa se está listo para armar los circuitos que, hasta ese momento, han sido sólo diagramas electrónicos. Esto suele hacerse en el la-

boratorio con un equipo especial que permite fácil interconexión entre circuitos integrados y otros elementos electrónicos y que no requiere soldadura (y, por tanto, resulta sencillo hacer cambios). En estas tarjetas especiales se construyen poco a poco los módulos que fueron diseñados en el papel, con lo que en corto tiempo se tiene un circuito real de prueba, como el que aparece en la foto, y que sirve para obtener la versión final del diagrama electrónico.

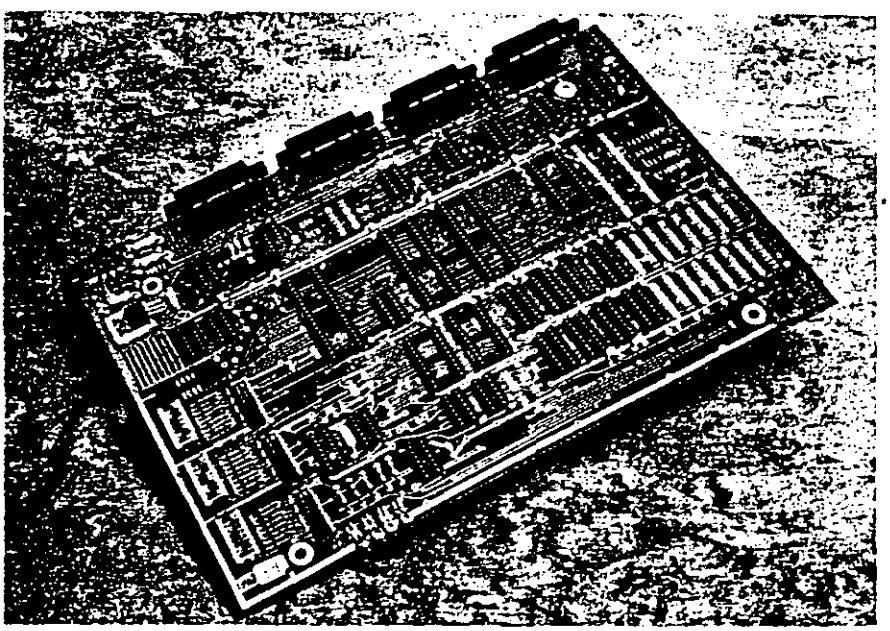


En paralelo con esta tarea se escriben los programas en lenguaje ensamblador para controlar el procesador de la anémona, y que residirán en una memoria no volátil, PROM. Una vez depurados los programas, se graban en el circuito PROM mediante un aparato especial, y el circuito se inserta en el lugar correspondiente dentro de la tarjeta de prueba. Se diseña también software de apoyo que se emplea para probar el funcionamiento de los dispositivos que conforman la arquitectura, porque no es posible probar el desempeño de este tipo de circuitos únicamente con mediciones eléctricas. En este momento se emplean también emuladores configurables que permiten hacer pruebas preliminares.

El circuito impreso

Con el conjunto completo de circuitos, los ingenieros están en posibilidad de hacer pruebas exhaustivas sobre el diseño total, y se llega así a la siguiente fase, que consistirá en la creación de una base definitiva para los circuitos,

empleando lo que se conoce como circuito impreso, en el que los alambres son reemplazados por pistas de cobre impresas sobre una tarjeta de material aislante. El dibujo del circuito impreso es una tarea muy especializada; culmina con un original que es enviado a un proceso fotográfico donde se reduce y queda listo para obtener de allí, masivamente, las tarjetas de circuito impreso. En las siguientes fotografías se muestra un ejemplo del dibujo final del diagrama electrónico, y de la tarjeta resultante.

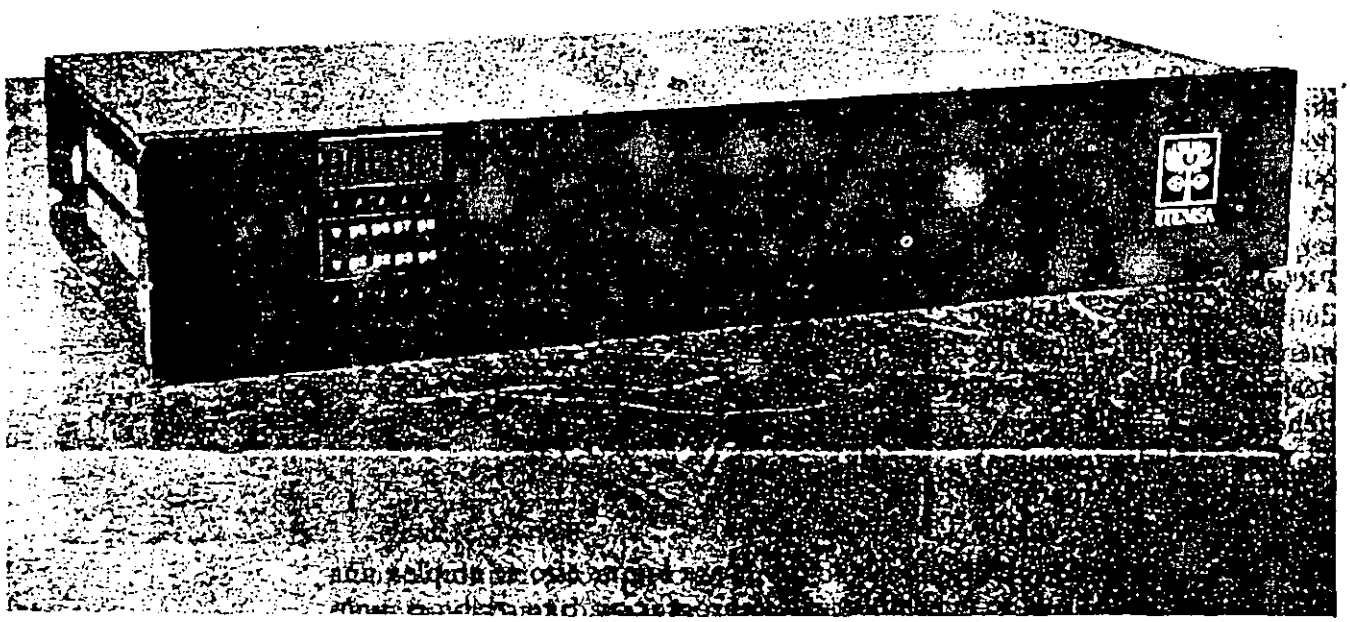


Cuando llegan las primeras muestras de los circuitos impresos, se montan y sueldan los componentes electrónicos y las memorias PROM sobre ellas, se obtiene entonces un primer prototipo del circuito terminado, listo para más pruebas. Esta fase de depuración es muy compleja, porque no es tan sencillo determinar si las fallas en el comportamiento esperado se deben al hardware o al software. Muchas veces sucede que se detecta la necesidad de hacer pequeñas alteraciones en algunas de las pistas del circuito impreso, y esto suele resolverse, si son pocas, mediante "puentes" formados con pequeños alambres que unen puntos de la placa o tarjeta. Los cambios requeridos en la memoria PROM son más sencillos de realizar, ya que simplemente se saca el circuito, se borra el programa que contiene (que está codificado en lenguaje de máquina) mediante una lámpara de luz ultravioleta y se vuelve a grabar con el nuevo programa (de hecho, para lograr esto se emplea una variante de PROM que se puede borrar y reprogramar, que recibe el nombre de EPROM). Que las alteraciones en la tarjeta sean demasiadas obliga

a un rediseño del circuito base para volver a obtener un original para producción, con los costos asociados. Normalmente se requieren entre dos y cinco revisiones del circuito para llegar a una versión por completo terminada y operativa. En el caso de la anémona fueron necesarias tres. En la siguiente foto se muestra una placa terminada con los componentes ya montados.

Primer prototipo para producción

A estas alturas del proyecto, cuando ya han transcurrido algunos meses desde la idea inicial, ya deben estar listos una buena cantidad de detalles (algunos de los cuales no son propiamente electrónicos) que forman parte indispensable de todo proyecto que tenga visos de distribución o comercialización, entre los que destacan la caja o chasis del aparato, que debe estar diseñado para que sea funcional y atractivo, la fuente de potencia, el manual para el usuario, el manual técnico de mantenimiento y un plan de presentación, mercadotecnia y distribución. Además, existe una serie de normas nacionales e internacionales (cuando se piensa en la exportación) que los aparatos electrónicos deben cumplir, en los campos de seguridad para el usuario, aislamiento contra fugas eléctricas, calor y ruido, y emisión de interferencia electromagnética. Los aspectos estéticos y mecánicos del producto final no deben subestimarse porque son complejos y requieren una fuerte inversión de recursos y tiempo.



Cortesía del Grupo Micrológica

Cuando el prototipo para producción está listo, se manda a algunos clientes potenciales para que hagan sus observaciones y sugerencias, que resultan muy valiosas porque surgen del uso real que se dará al producto.

Muchas veces sucede que son estos aspectos los que retrasan la última fase de creación de un producto: la producción industrial. Una cosa es obtener un prototipo terminado, y otra muy diferente es producir centenares o miles de esos aparatos en una fábrica o taller de montaje. Para esto se requiere un plan completo de producción, que comienza con enviar a la fábrica todos los diagramas (eléctricos, electrónicos, de montaje y de diseño final) junto con la lista de especificación de las partes (y sus posibles sustitutos) y las guías de ensamble y las pruebas para el control de calidad. En ocasiones se vuelve necesario desarrollar sistemas de prueba casi tan complicados como el producto mismo, sobre todo cuando la producción es masiva. Ya no se hablará aquí de la fase de producción, porque esto es un tema de ingeniería industrial, que queda fuera de las tareas de un laboratorio de diseño de prototipos como el aquí ilustrado.

Palabras y conceptos clave

En esta sección se agrupan las palabras y conceptos de importancia que se estudiaron en el capítulo, con el objetivo de que el lector pueda hacer una autoevaluación que consiste en ser capaz de describir con cierta precisión lo que cada término significa, y no sentirse satisfecho hasta haberlo logrado. Se muestran en el orden en que se describieron o se mencionaron.

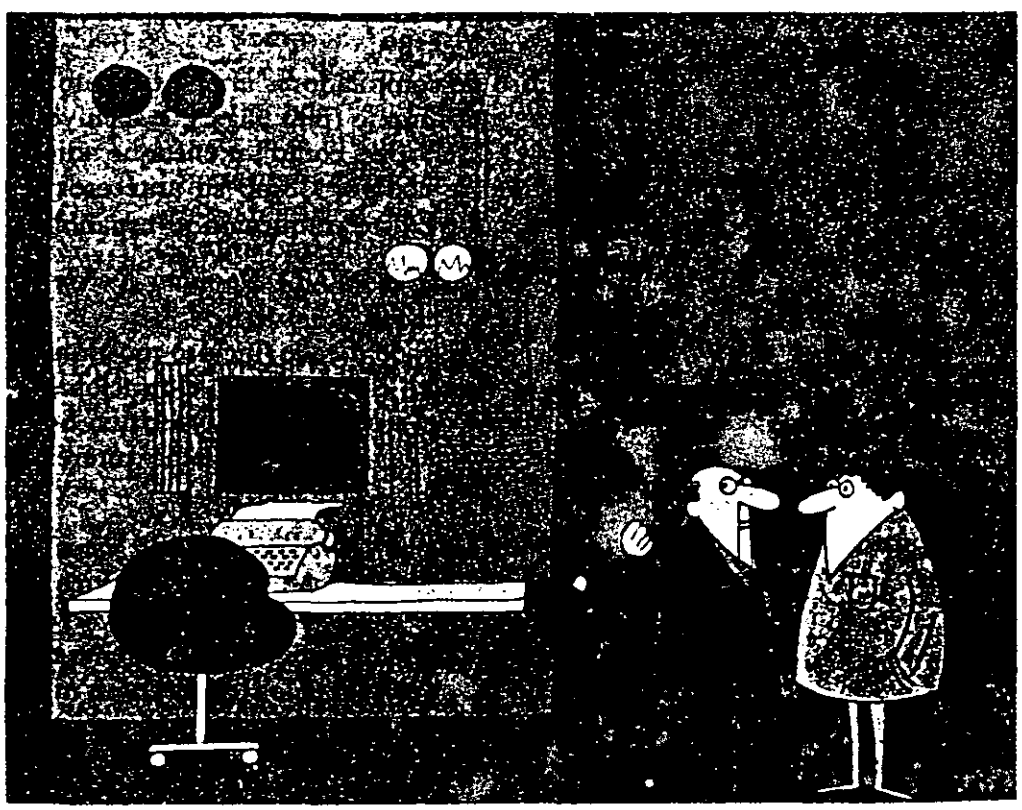
PROGRAMACIÓN DE SISTEMAS
MODOS DE DIRECCIONAMIENTO
DIRECCIÓN ABSOLUTA
REFERENCIA SIMBÓLICA
VARIABLE
ETIQUETA
ENSAMBLADOR
TABLA DE SÍMBOLOS
MACROPROCESAMIENTO
MACROS
PARÁMETROS
MACROENSAMBLADOR
CARGADOR
BOOTSTRAP
IPL
PROGRAMA MONITOR
ESTRUCTURA DEL LENGUAJE
ANÁLISIS LÉXICO
ANÁLISIS SINTÁCTICO
ANÁLISIS SEMÁNTICO
COMPILADOR
COMPONENTE LÉXICO *(TOKEN)*
GRAMÁTICA
GENERACIÓN DE CÓDIGO
OPTIMIZACIÓN
INTÉRPRETE
COMPILADOR DE COMPILADORES
SISTEMA OPERATIVO
NÚCLEO DEL SISTEMA *(KERNEL)*
PROGRAMA (CONJUNTO DE INSTRUCCIONES)
PROCESO (CONJUNTO DE ACCIONES)
PROCESADOR
CONCURRENCIA
SIMULTANEIDAD
MULTIPLEXACIÓN EN TIEMPO
OPERACIÓN PRIVILEGIADA
DESPACHADOR
VECTOR DE ESTADO (PSW)
ESTADOS DE UN PROCESO
MULTIPROGRAMACIÓN
MANEJO DE MEMORIA POR PARTICIONES
FRAGMENTACIÓN
RELOCALIZACIÓN
SWAPPING
PAGINACIÓN
MEMORIA VIRTUAL
MEMORIA *CACHÉ*
UNIDAD DE MANEJO DE MEMORIA (MMU)
SEGMENTACIÓN
SCHEDULER
TIEMPO COMPARTIDO
SPOOL
SISTEMA DE ARCHIVOS
LENGUAJE DE CONTROL
EDITOR DE PANTALLA
EDITOR DE LÍNEA
BASE DE DATOS
ESQUEMA DE LA BASE DE DATOS
LENGUAJE DE CONSULTAS *(QUERY LANGUAGE)*
MODELOS DE DATOS
HOJA DE CÁLCULO
INTELIGENCIA ARTIFICIAL
SISTEMA EXPERTO
CIRCUITO IMPRESO
DESENSAMBLADOR

Ejercicios

1. Haga un resumen de las operaciones del lenguaje de máquina de la computadora empleada en algún centro de cómputo que le sea accesible y compárelas contra las expuestas al inicio del capítulo.
2. ¿Puede existir un procesador que tenga la instrucción
   `INV_MAT N, M`
   que invierte una matriz de orden $N \times M$, como parte de su lenguaje de máquina? ¿Qué implicaciones tendría esta posibilidad?
3. En realidad, un ensamblador no requiere dar un segundo paso completo sobre el texto del programa fuente, sino que basta con que almacene la posición, dentro del programa fuente que se está ensamblando, de los identificadores (variables o etiquetas) a los que se hace referencia pero que aún no han sido definidos. Para esto es suficiente con una tabla de referencias pendientes. Describa con el mayor detalle posible cómo lograr esto para obtener un ensamblador de "un paso y medio".
4. Un **desensamblador** es un programa que lee un archivo que contiene un programa objeto y reconstruye el programa fuente original escrito en lenguaje ensamblador. En principio, esto es posible porque existe una correspondencia uno a uno entre las instrucciones del lenguaje de máquina y los mnemónicos de un lenguaje ensamblador, como se ha explicado. Diseñe un desensamblador, con el mismo nivel de detalle que el empleado en la sección 4.2 del texto.
5. ¿Puede un desensamblador reconstruir los nombres simbólicos de las etiquetas de un programa? Justifique su respuesta.
6. ¿Puede una macrodefinición lograr resultados que no estén considerados dentro de las capacidades directas del lenguaje de máquina? Justifique su respuesta.
7. Encuentre y analice ejemplos del problema del *bootstrap* en la vida cotidiana (hay muchos). He aquí uno de ejemplo: la Asociación de Ciudadanos Distinguidos decide, mediante consenso de su comité de admisiones, quiénes pueden adquirir la categoría de ciudadano distinguido; ¿quién nombró inicialmente al comité?
8. ¿Puede un programa escrito en un lenguaje de programación compilable lograr resultados que no estén considerados en las capacidades directas del lenguaje de máquina? Justifique su respuesta.
9. Un compilador, a fin de cuentas, no es más que un programa que debe estar escrito en algún lenguaje. ¿Puede el compilador del lengua-

je "A" estar escrito en el lenguaje "A"? Analice con cuidado las implicaciones de su respuesta.

10. ¿Puede existir un "descompilador"? Justifique su respuesta.
11. ¿A qué se debe que algunas microcomputadoras "entienden" BASIC (por ejemplo) al momento de encenderlas, sin que el usuario tenga que hacer nada más? ¿Será BASIC el lenguaje de máquina de esos procesadores? Explique su respuesta.
12. En el texto se menciona que un sistema operativo es un gran conjunto de programas, que rebasa las capacidades de una sola persona para entenderlo completamente. Sin embargo, el sistema operativo que fue estándar en las primeras microcomputadoras, llamado CP/M, fue producido por un solo especialista, Gary Kildall, que lo comenzó a vender en forma particular en 1974. ¿Cómo es posible esto?
13. La política de admisiones empleada en algunos restaurantes que consiste en sólo asignar mesas pequeñas a clientes que llegan solos o en pareja parece adecuada en general. Sin embargo, no siempre es la mejor. Considere el caso de que todas las mesas pequeñas están ocupadas y no se deja entrar a una persona sola, siendo que hay varias mesas grandes libres. ¿Cuál sería el tiempo mínimo que debería esperarse antes de dejarlo pasar? ¿Qué pasa si se le permite pasar e inmediatamente después llega un grupo de varias personas? Diseñe diversas políticas de asignación de mesas para el ejemplo anterior y compárelas con lo que el texto menciona en la sección de manejo de memoria y



"De vez en cuando la desconecto para que sepa quién es el que manda".

de procesador, suponiendo que las mesas son las áreas de memoria, los clientes son los procesos y el encargado de las admisiones es el *scheduler*.

14. El señor K hace una reservación para un vuelo en una oficina de una aerolínea en su ciudad. Él cree que la terminal en la que se teclearon sus datos es la computadora, pero en realidad no es más que la terminal de video. En muchos casos, la computadora no está en la misma ciudad y a veces ni siquiera en el mismo país. Describa, lo más estructuradamente posible (empleando el modelo de sistemas operativos descrito en el capítulo), los pasos necesarios para que la computadora remota almacene los datos del señor K.

Referencias para el capítulo 4*

[AHOA77] Aho, Alfred y Jeffrey Ullman, *Principles of Compiler Design*, Addison-Wesley, Massachusetts, 1977.
Libro extremadamente amplio y complejo que estudia técnicas establecidas para el diseño de compiladores, junto con nuevos desarrollos de la teoría matemática de la computación aplicada a los lenguajes de programación y sus procesadores. Se trata, sin duda, de un libro avanzado.

[AHOA85] Aho, Alfred, Ravi Sethi y Jeffrey Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Massachusetts, 1985.
Nueva versión del libro de compiladores arriba descrito. Ahora el texto es aún más amplio (800 págs.), e incluye nuevas técnicas para el diseño, así como un enfoque adicional en la generación de código. Como su antecesor, éste es un texto avanzado. Existe traducción al español.

[BACM86] Bach, Maurice, *The Design of the UNIX Operating System*, Prentice-Hall, New Jersey, 1986.
En este libro se describe con todo detalle (incluso mostrando programas reales escritos en el lenguaje C) el funcionamiento interno y el diseño del sistema operativo Unix. Se trata de un libro de nivel avanzado, de mucho interés para el especialista.

[BECL88] Beck, Leland, *Software de sistemas: Introducción a la programación de sistemas*. Addison-Wesley Iberoamericana, México, 1988.
Traducción del libro más reciente sobre el tema general de la programación de sistemas. Trata los temas en un ni-

* En términos generales (excepto los artículos mencionados) estas referencias son de nivel avanzado, puesto que la programación de sistemas es un área de estudio especializada. Algunos de estos libros son propios para estudios de posgrado en computación, mientras que otros se emplean generalmente en las licenciaturas en computación o ingeniería electrónica.

vel introductorio, pero tiene la ventaja de que trabaja sobre una computadora "de papel" diseñada ex profeso para propósitos académicos, y los ejemplos del texto usan ese lenguaje de máquina. También tiene algunas descripciones de sistemas reales.

[BENA82] Ben-Ari, M., *Principles of Concurrent Programming*, Prentice Hall International, Londres, 1982.
Uno de los pocos libros dedicados al tema especializado de la programación concurrente, que cada vez adquiere más importancia. Existen varios lenguajes de programación concurrente (junto con sus respectivos compiladores) que sirven para simular directamente la ejecución de procesos en paralelo, y este libro contiene el diseño de un compilador para el lenguaje Pascal concurrente.

[BERH80] Berliner, Hans, "Computer Backgammon", en *Scientific American*, junio, 1980.
El autor del programa que le ganó al campeón mundial de backgammon explica sus puntos de vista sobre la inteligencia artificial y la forma en que las computadoras manejan juegos de estrategia. En un número anterior de la misma revista se dedicó un artículo a un programa para jugar póquer.

[BEVT85] Thompson, Beverly y William Thompson, "Inside an Expert System", en *Byte*, abril, 1985.
Muy interesante artículo que muestra, con cierto detalle técnico, el funcionamiento de una máquina de inferencias para procesar información con base en el significado y aplicación de reglas predefinidas, que forman parte de una base de conocimientos. Se muestran ejemplos de algunos de los componentes de los programas que forman un sistema experto.

[BRIH73] Brinch Hansen, Per, *Operating Systems Principles*, Prentice Hall, New Jersey, 1973.
Éste es uno de los pocos libros sobre sistemas operativos que los abordan desde un punto de vista matemático, a diferencia de otros (Donovan, Lister y Deitel, por ejemplo) que les dan un tratamiento más bien pragmático. El autor ha hecho trabajos muy importantes sobre lenguajes para manejar concurrencia.

[BROP75] Brown, P. J., *Macro Processors*, Wiley and Sons, Londres, 1975.
Descripción muy amplia y bien documentada sobre los sistemas de macroprocesamiento, junto con ejemplos reales de su uso. Se analizan temas avanzados sobre software compatible y portable.

[DATC86] Date, C. J., *Introducción a los sistemas de bases de datos*, Addison-Wesley Iberoamericana, México, 1986.
Traducción de la tercera edición de un conocido libro sobre diseño y uso de bases de datos. Se describen las características generales de los sistemas manejadores de bases de datos junto con consideraciones teóricas sobre diseño de esquemas.

[DEIH87] Deitel, Harvey, *Introducción a los sistemas operativos*, Addison-Wesley Iberoamericana, México, 1987.
Traducción de un libro reciente sobre sistemas operativos. Es gran volumen (700 págs.), y en él se describe el funcionamiento de cada parte de un sistema operativo en un nivel conceptual. Se dedican varios capítulos a las características de algunos sistemas operativos reales.

[DENP71] Denning, Peter, "Third Generation Computer Systems", en *Computing Surveys*, Association for Computing Machinery, vol. 3, núm. 4, diciembre, 1971.
Excelente artículo sobre los sistemas operativos de las computadoras que manejan multiprogramación, memoria virtual y otros adelantos propios de la tercera generación. Aunque no es reciente, las ideas y conceptos aquí explicados mantienen su actualidad y están tratados en un nivel bastante comprensible.

[DENP84] Denning, Peter y Robert L. Brown, "Operating Systems", en *Scientific American*, septiembre, 1984.
Breve introducción a las funciones de los sistemas operativos que divide su desempeño en 13 niveles de operación, desde los circuitos electrónicos hasta la comunicación con el usuario; esto sirve para dar una idea somera de su estructuración jerárquica. Existe traducción al español.

[DEWA84] Dewdney, A. K., "Computer Recreations", en *Scientific American*, julio, 1984.
Artículo que explica algunos de los procesos mediante los cuales una computadora puede jugar "damas", y aprovecha para hacer consideraciones sobre el futuro de la inteligencia artificial. Esta sección de la revista a cargo del Sr. Dewdney se dedica a presentar mes con mes pasatiempos y consideraciones de muchísimo interés sobre la teoría de la computación y las computadoras.

[DONJ72] Donovan, John, *Systems Programming*, McGraw-Hill International, Tokio, 1972.
Éste es un libro que tiene algunos capítulos excelentes, junto con otros de relativamente poco interés, sobre todo por estar demasiado enfocado a los manejos internos de

la serie IBM 360. Durante muchos años fue prácticamente el único libro sobre programación de sistemas en general. Existe traducción al español.

[DOUJ78] Douglas, J. R., "Chess 4.7 versus David Levy: the computer beats a Chess Master", en *Byte*, diciembre, 1978. En 1968 el maestro internacional de ajedrez David Levy hizo una apuesta pública de mil libras esterlinas "contra cualquier computadora que me derrote en un torneo de ajedrez en los próximos diez años". Este artículo narra las peripecias del último intento que hizo la computación por derrotar al maestro inglés, justo en 1978. La máquina perdió el torneo 1 1/2 contra 3 1/2, lo que significa que fue capaz de ganar un juego y empatar otro. Véase también el artículo "An Advice Taking Chess Computer", de Albert Zobrist y Frederic Carlson, publicado en la revista *Scientific American*, junio de 1973. Tal vez pronto llegue el día en que la inteligencia artificial vuelva por sus fueros.

[GRAR88] Grauer, Robert y Paul Sugrue, *Aplicaciones de microcomputadoras*, McGraw-Hill, México, 1988.
Traducción de un compendio de los principales campos de aplicación de las computadoras personales, en el que se describe el funcionamiento y las características generales de los programas comerciales más difundidos en las áreas de procesamiento de textos, hojas electrónicas de cálculo, manejadores de bases de datos y, finalmente, sistemas de comunicaciones y el concepto de software integrado.

[GRID71] Gries, David, *Compiler Construction for Digital Computers*, Wiley, Nueva York, 1971.
Buen libro sobre compiladores, aunque, por haber sido escrito hace varios años, no considera los nuevos algoritmos y métodos de análisis sintáctico ascendente. El autor publicó hace poco un texto sobre la ciencia de la programación.

[HOPJ69] Hopcroft, John y Jeffrey Ullman, *Formal Languages and their Relation to Automata*, Addison-Wesley, Massachusetts, 1969.
Libro de nivel superior que describe lo que hasta 1969 se sabía sobre las matemáticas computacionales. Diez años más tarde los autores escriben otro volumen, que contiene nuevos desarrollos. Aquí se estudian a fondo temas sobre lenguajes y gramáticas formales, autómatas y reconocedores, máquinas de Turing y teoría de la computabilidad.

[HOPJ79] Hopcroft, John y Jeffrey Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Massachusetts, 1979.
Este libro comienza diciendo: "Hace diez años los autores produjeron un libro que cubría el material conocido sobre lenguajes formales, teoría de autómatas y complejidad computacional. Visto restrospectivamente, sólo algunos resultados importantes se omitieron en sus 237 páginas. Al escribir un nuevo libro sobre estos temas nos encontramos con que el campo se ha ampliado en tantas nuevas direcciones que es imposible tratarlas todas de manera exhaustiva".

[KERB76] Kernighan, Brian y P. J. Plauger, *Software Tools*, Addison Wesley, Massachusetts, 1976.
Los autores, que han participado activamente en el desarrollo de nuevos lenguajes de programación (como C) y nuevos sistemas operativos (como Unix), proponen en este libro su filosofía computacional, consistente en diseñar herramientas de programación para auxiliarse en la construcción de sistemas complejos.

[LISM85] Lister, M. A., *Fundamentals of Operating Systems*, Macmillan, Londres, 1985.
Tercera edición de una excelente descripción de un sistema operativo "de papel", tratado en nivel intermedio. Incluye prácticamente todos los temas de estudio y la nueva edición incorpora un capítulo sobre medidas de desempeño y funcionamiento. Se recomienda para un primer curso de este tema.

[MADS74] Madnick, Stuart y John Donovan, *Operating Systems*, McGraw Hill, Nueva York, 1974.
Se puede considerar a este libro como uno de los pioneros en los textos sobre sistemas operativos. Está tratado desde una perspectiva entre lo descriptivo y lo operativo, y varios de sus capítulos siguen siendo vigentes. Contiene un ejemplo completo de un núcleo codificado en ensamblador IBM 370. Existe traducción al español.

[MORC84] Morgan, Christopher y Mitchell Waite, *Introducción al microprocesador 8086/8088*, Byte/McGraw-Hill, México, 1984.
Traducción de un libro sobre manejo y programación de los microprocesadores de la familia Intel 8086 y 8088 de 16 bits. Se trata de un libro técnico, dirigido a lectores que ya manejan los conceptos de programación en lenguaje de máquina y en ensamblador. Dedica capítulos a los procesadores aritmético y de entrada/salida de la misma familia.

[PAVR81] Pavelle, Richard, M. Rothstein y J. Fitcher. "Computer Algebra", en *Scientific American*, diciembre, 1981.
Artículo sobre los sistemas computacionales para manejo algebraico. Tiene varios ejemplos de las transformaciones y simplificaciones que se logran, y muestra algunos de los métodos con los que funcionan. Existe traducción al español.

[PETJ83] Peterson, James y Abraham Silberschatz, *Operating System Concepts*, Addison-Wesley, Massachusetts, 1983.
Buen libro sobre sistemas operativos que describe, con un adecuado nivel de detalle, múltiples conceptos tanto teóricos como prácticos, e incluye ejemplos particulares de cada tema. Resulta una excelente y amplia fuente de estudio global.

[SCIA71] Scientific American, *Computers and Computation*, W. H. Freeman, San Francisco, 1971.
Aunque este libro es de hace varios años, es una excelente compilación de artículos de interés general sobre computación que esta prestigiada revista publicó en la década de 1970. Tiene algunos artículos considerados como clásicos. Existe traducción al español.

[TANA87] Tanenbaum, Andrew, *Operating Systems: Design and Implementation*, Prentice-Hall, New Jersey, 1987.
Libro de nivel superior sobre sistemas operativos. Además de ser nuevo, resulta muy interesante porque contiene el listado completo de un sistema operativo de fines didácticos escrito en lenguaje C y basado en la filosofía de Unix. Aunque el sistema mostrado se llama MINIX (porque es una especie de Unix en miniatura), ocupa más de 250 páginas en un apéndice del libro.

[TREJ85] Tremblay, Jean-Paul y Paul G. Sorensen, *The Theory and Practice of Compiler Writing*, McGraw-Hill, Nueva York, 1985.
Uno de los libros de texto más nuevos sobre compiladores. Extremadamente amplio (800 págs.), cubre todos los temas de esta materia y explora multitud de técnicas de análisis y de generación de código. Existe un anexo con la codificación de un compilador completo escrita en el lenguaje PL/I.

[ULLJ76] Ullman, Jeffrey, *Fundamental Concepts of Programming Systems*, Addison-Wesley, Massachusetts, 1976.
Excelente revisión de la programación de sistemas (no incluye los sistemas operativos), por uno de los más autori-

zados y prolíficos autores de la literatura computacional de primer nivel.

[ULLJ82] Ullman, Jeffrey, *Principles of Database Systems*, Computer Science Press, Maryland, 1982.
Libro de teoría matemática de bases de datos. A diferencia de casi todos los demás textos sobre bases de datos, éste se dedica a la fundamentación formal de las propiedades de los diferentes modelos (jerárquico, de red, relacional), y hace uso de teoremas y demostraciones, además de tratar con rigor temas diversos, como las bases de datos distribuidas.

[WIEG83] Wiederhold, Gio, *Diseño de bases de datos*, McGraw-Hill, México, 1983.
Traducción de la segunda edición de un amplio libro sobre sistemas de bases de datos. Tiene toda una sección sobre manejo de archivos y contiene también análisis matemáticos de la eficiencia de los esquemas propuestos.

[WINP84] Winston, Patrick, *Artificial Intelligence*, Addison-Wesley, Massachusetts, 1984.
Segunda edición del libro considerado como estándar en el campo de la inteligencia artificial. El autor también es ampliamente conocido por sus trabajos con el lenguaje de programación LISP (entre los que se cuenta un excelente libro sobre el tema), y en esta segunda edición intenta cubrir lo nuevo sobre visión y principios de adquisición de conocimiento. El prefacio inicia diciendo: "El campo de la inteligencia artificial ha cambiado enormemente desde la primera edición de este libro", y el lector debe tomar en cuenta que se refiere tan sólo a un periodo de menos de quince años.

[WINT84] Winograd, Terry, "Computer Software for Working with Language", en *Scientific American*, septiembre, 1984.
En este artículo se muestra cómo una computadora puede explorar el significado de una frase de lenguaje natural, entrando a los temas de procesamiento de textos, análisis sintáctico y semántico y problemas asociados. El artículo concluye con la observación de que "el software para simular el entendimiento completo del lenguaje simplemente no está en consideración". Existe traducción al español.

Segunda parte

5

Computabilidad

5.1 Introducción

> Dado que va a pasar no sé qué no sé cuándo, ¿qué disposiciones habrá que tomar?
>
> *Jean Tardieu*
>
> "Problemas y trabajos prácticos",
> en la revista *El Cuento*, núm. 13, junio de 1965.

Tal vez la idea central de la computabilidad (que es el término matemático con que se conocen los estudios sobre teoría de la computación) consista en ser capaz de encontrar la representación adecuada para la descripción de un problema o de un fenómeno.

Es evidente que siempre será posible describir algún aspecto de la realidad por medio de cierto lenguaje; basta con encontrar las combinaciones adecuadas de símbolos para representar lo que se tiene en mente. El concepto de símbolo es, pues, fundamental en cuanto a la habilidad para describir que por excelencia tiene el ser humano. Buena parte de nuestros procesos mentales y psicológicos se reducen a descripciones que hacemos respecto a la realidad que nos rodea, para lo cual se requieren asociaciones entre conceptos y elementos del lenguaje (símbolos) en su sentido más amplio.

Pero una vez resuelto, aunque sea en principio, el problema de poder describir el mundo, hay que definir la manera de confirmar si la descripción es completa. Es decir, ¿cómo asegurar que una descripción puede ser reproducida por un tercero, para que éste llegue al mismo lugar del que partimos?

Este es otro problema: representar el fenómeno descrito. La comunicación efectiva tiene lugar cuando se describe un problema o fenómeno determinado ante un receptor, y éste desarrolla la descripción que se emitió y vuelve al objeto original. Si se logra cerrar el círculo descripción-representación, entonces se está hablando de un conocimiento transmisible.

El ciclo descripción-representación

Tal vez este análisis parezca extraño para quien está acostumbrado a pensar en una computadora como una gran y veloz calculadora. Un poco más adelante se verá que una máquina de esta clase es en realidad un modelo general, que permite cerrar el círculo arriba enunciado y que, por ende,

puede perfectamente simular una calculadora. Lo importante ahora es comprender que la computadora es mucho más que una calculadora compleja, puesto que está basada en una idea matemática mucho más potente, la del modelado.

El concepto de modelo

Un modelo es una especificación, generalmente en términos de un lenguaje matemático, de los pasos necesarios para reproducir, aquí y ahora, un subconjunto determinado de la realidad descrito previamente. Es más, un modelo parte siempre de la descripción de lo que se representará.

Surge una pregunta: ¿Todo aquello que es descriptible será representable? Es decir, ¿se podrá siempre pasar de la descripción de un proceso a su representación? Intuitivamente, parece que sí; sólo hay que asociar acciones a símbolos de la descripción y se estará "actuando" la descripción que nos fue dada. Pero —nos preguntamos—, ¿esta representación simulará completamente lo que fue descrito? En apariencia, la exactitud de la simulación depende de la exactitud de la descripción. Cuanto más adecuada sea la descripción del proceso, tanto mejor será el resultado que emula lo real.

Descubriremos más adelante que esta idea es incorrecta; esto es, existen ciertos procesos que pueden ser descritos con un grado ilimitado de precisión, pero cuya representación fracasa; no llega de regreso al punto de partida.

Tal vez si se replantea el problema se alcance una mejor comprensión. Supóngase que se fabrica un aparato para producir descripciones en términos de cadenas de símbolos. Supóngase también que se desea construir una máquina que se comporte de la siguiente manera: dada una descripción cualquiera, la analizará durante un tiempo finito y después emitirá su dictamen, que consistirá en un sí o un no. El sí querrá decir que el fenómeno descrito es representable, y el no indicará lo contrario. Una respuesta positiva significará, por ejemplo, que el problema que está siendo actuado tiene una solución efectiva, mientras que la respuesta negativa significará que el problema no tiene solución.

Un problema de esta clase es, por ejemplo: dado un número, extraer su raíz cuadrada. El aparato descriptor produce una cadena de símbolos que, una vez en la máquina, provoca que ésta dictamine sí o no.

Está claro que la máquina funcionará como un **procedimiento de decisión**. Decidirá, en un tiempo finito, si la descripción es representable o no y, por tanto, si el problema asociado con ella tiene o no solución.

El problema de la decisión

Ahora viene una pregunta mucho más comprometedora, ¿existe una máquina como ésta? (el hecho de que se pueda idear no significa que exista). Nos estamos acercando ya al problema central de la **teoría de la computabilidad**, que es precisamente el de encontrar maneras de representar descripciones de procesos, de manera tal que siempre se pueda decir sí (existe) o no.

Se dice que un problema es *computable* cuando existe una de estas máquinas de decisión para él. Entonces la pregunta podría plantearse como sigue: ¿todos los procesos son computables?

La respuesta, asombrosamente, es *no*.

Para demostrarlo, habría que encontrar al menos un proceso así. O sea, encontrar una descripción tal que cuando sea alimentada a la máquina que

decide, la ponga en un estado curioso: no dice que sí, pero tampoco dice que no. O sea, ¡no puede decidir!

Este resultado, que parece contradecirse con la intuición, se debe al matemático inglés Alan Mathison Turing (1912-1954), quien en el proceso de encontrar esa máquina descubrió propiedades insospechadas acerca del mundo de los conceptos lógico-matemáticos.

Alan M. Turing

El sistema de pensamiento formal tiene limitantes

Como resultado de las investigaciones de Turing ahora se conocen algunas limitantes, inherentes a nuestro sistema de pensamiento, de los mecanismos mentales para averiguar la estructura formal del mundo. Estos estudios inauguran la **teoría matemática de la computación**, de la cual las computadoras son tan sólo un aspecto, el más visible sin duda.

Pero volvamos a nuestro tema de estudio, los algoritmos. Diremos que un **algoritmo** es una manera formal y sistemática de representar la descripción de un proceso. Se ha dicho ya que no todos los procesos terminan, es decir, que no para todos los casos se puede decidir si cierta descripción es representable o no.

5.2 El concepto de algoritmo: la máquina de Turing*

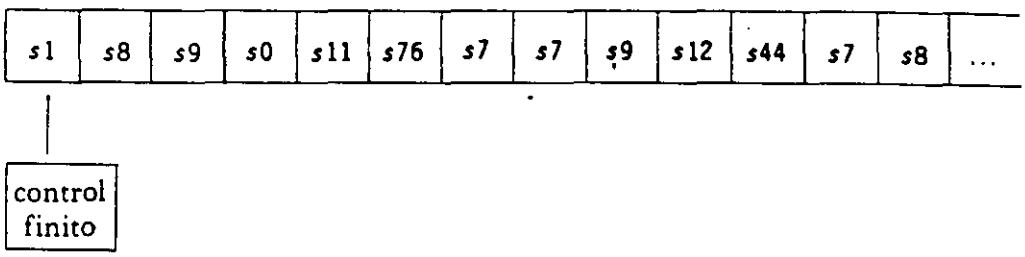
En su estudio, Turing propone una manera para representar un proceso dada su descripción. El modelo matemático propuesto (conocido ahora como

* Esta "máquina" no debe confundirse con un aparato físico o mecánico. Se trata más bien de una construcción matemática, y no tiene nada que ver con motores, transistores o dispositivos de ningún tipo.

Máquina de Turing consta de los siguientes elementos: una cinta de longitud infinita, dividida en celdas (cada celda puede contener un símbolo, tomado de un diccionario de símbolos predefinido), y un control finito, que tiene la capacidad de examinar algún símbolo de alguna celda y tomar una decisión, que depende del símbolo observado y del estado en que se encuentre el control finito. (Para un estudio de estos temas, en general avanzados, véase el artículo [HOPJ84] y la referencia [HOPJ79] del capítulo 4.)

El control se llama *finito* precisamente porque puede, para un momento determinado, estar en uno de varios estados posibles, habiendo tan sólo un número finito de ellos. Se puede pensar en un estado como una configuración de una máquina discreta que, según se dijo en el capítulo 1, tiene la capacidad de adoptar un estado u otro, pero ninguno intermedio.

Si se supone un diccionario de símbolos [$s1$, $s2$, ..., sn], podemos pensar en codificar un proceso (describirlo) por medio de ellos, para luego escribir estos símbolos —uno a uno— en celdas de la cinta de la máquina de Turing, que entonces se vería así:



En esta figura, el control finito de la máquina de Turing se encuentra en la primera celda de la cinta, esto es, observando el primero de los símbolos que describe el proceso que se desea representar; y adopta cierto estado, que recibe el nombre de estado inicial.

Ahora la máquina se comportará emulando a un ser vivo, en el sentido de que reaccionará, de maneras preestablecidas, ante estímulos que recibe del mundo exterior (que, para la máquina, estará representado por la cinta).

Definiremos un estímulo como la conjunción de dos sucesos: 1) que el control finito se encuentre en cierto estado $e1$ justo cuando 2) en la celda que está siendo observada hay un símbolo $s1$.

Ante este estímulo la máquina tendrá una reacción, que consistirá en tres cosas:

Modo de funcionamiento de una máquina de Turing

1) pasará a un nuevo estado (aunque este nuevo estado puede perfectamente ser igual al que tenía antes, lo que es equivalente a que no cambie),
2) escribirá un nuevo símbolo en el lugar del recién leído (que también puede ser el mismo que antes) y,
3) moverá el control finito una celda a la derecha (D) o una celda a la izquierda (I).

Ahora es posible escribir, para una máquina en particular, una conjunto de reacciones que sucederán cuando se presente un conjunto de estímulos:

← ESTÍMULOS →		← REACCIONES →		
Estado actual	Símbolo actual	Estado nuevo	Símbolo nuevo	Movimiento
*e*15	*s*38	*e*21	*s*42	I
*e*21	*s*42	*e*21	*s*11	D
*e*09	*s*14	*e*42	*s*14	D
*e*09	*s*13	*e*15	*s*82	I
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

En esta tabla la primera entrada se lee: "si cuando la máquina se encuentra en el estado 15 el control se halla observando el símbolo 38, pasará entonces al estado 21, escribirá en esa misma celda el símbolo 42, y se moverá una celda a la izquierda".

De manera similar se leen las demás entradas de la tabla. Observése que en el segundo renglón de la tabla la máquina no cambia de estado (porque, encontrándose en el estado 21, pasa al estado 21, o sea, al mismo). De la misma forma, en el tercer renglón la máquina deja el símbolo observado sin modificar, porque vuelve a escribir el que ya estaba.

Con un poco de imaginación podemos darnos cuenta de que es posible programar la máquina para que se comporte de tal forma que logre algún fin previsto. Tal fin no será otro más que, precisamente, representar (darle vida o actuar) la descripción de un proceso que fue adecuadamente codificado en la cinta. El resultado que la máquina dejará al término de su actuación será un conjunto de celdas de la cinta que contiene la codificación de la solución encontrada. Por último, es necesario definir cierto estado (o varios de ellos) como estado final, de modo que, cuando el control finito llegue a él (o a alguno de ellos), se detenga, dando así por terminada la computación del proceso.

En el siguiente ejemplo se describe la tabla de cierta máquina de Turing que llamaremos MT1; tiene siete estados ($e0, \ldots, e6$), y maneja seis símbolos ($0, 1, X, Y, Z, B$)

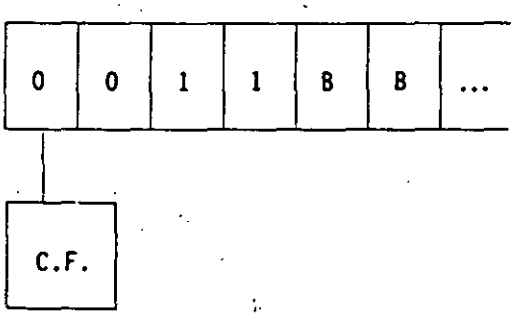
Estímulos		Reacciones		
e0	0	e1	X	D
e0	1	e6	1	D
e1	0	e1	0	D
e1	Y	e1	Y	D
e1	1	e2	Y	I
e2	Y	e2	Y	I
e2	X	e3	X	D
e2	0	e4	0	I
e3	Y	e3	Y	D
e3	B	e5	Z	I
e3	1	e6	1	D
e4	0	e4	0	I
e4	X	e0	X	D

Para esta máquina, los estados finales son $e5$ y $e6$, y el estado inicial es $e0$.

Originalmente la cinta contiene tan sólo ceros, unos y blancos ($0, 1, B$), en alguna configuración cualquiera, que puede cambiar de un caso a otro, o de un ejemplo a otro.

Las letras X, Y, Z son símbolos auxiliares que el modelo emplea para su funcionamiento.

Véase qué sucede cuando el control finito tiene enfrente una cinta con los siguientes símbolos y se encuentra observando el primero.



El lector puede comprobar que se presentará la siguiente secuencia de estados y de símbolos en la cinta. Se muestran los estímulos (estado actual-cinta actual) y a su derecha las acciones. Hay un asterisco debajo del símbolo que el control finito observa en cada momento:

Estado actual	Cinta actual	Cinta modificada
1) *e*0	0 0 1 1 *B* *B*	*X* 0 1 1 *B* *B*
2) *e*1	*X* 0 1 1 B B	*X* 0 1 1 *B* *B*
3) *e*1	*X* 0 1 1 B B	*X* 0 *Y* 1 *B* *B*
4) *e*2	*X* 0 *Y* 1 B B	*X* 0 *Y* 1 *B* *B*
5) *e*4	*X* 0 *Y* 1 B B	*X* 0 *Y* 1 *B* *B*
6) *e*0	*X* 0 *Y* 1 B B	*X* *X* *Y* 1 *B* *B*
7) *e*1	*X* *X* *Y* 1 *B* *B*	*X* *X* *Y* 1 *B* *B*
8) *e*1	*X* *X* *Y* 1 *B* *B*	*X* *X* *Y* *Y* *B* *B*
9) *e*2	*X* *X* *Y* *Y* *B* *B*	*X* *X* *Y* *Y* *B* *B*
10) *e*2	*X* *X* *Y* *Y* *B* *B*	*X* *X* *Y* *Y* *B* *B*
11) *e*3	*X* *X* *Y* *Y* *B* *B*	*X* *X* *Y* *Y* *B* *B*
12) *e*3	*X* *X* *Y* *Y* *B* *B*	*X* *X* *Y* *Y* *B* *B*
13) *e*3	*X* *X* *Y* *Y* *B* *B*	*X* *X* *Y* *Y* *Z* *B*
14) *e*5	*X* *X* *Y* *Y* *Z* *B*	fin

Veamos ahora cómo se comporta la máquina cuando se le presenta esta nueva cinta:

0	1	1	B	B	...

Estado actual	Cinta actual	Cinta modificada
1) *e*0	0 1 1 *B* *B*	*X* 1 1 *B* *B*
2) *e*1	*X* 1 1 *B* *B*	*X* *Y* 1 *B* *B*

3) *e*2	X Y 1 B B *	X Y 1 B B
4) *e*3	X Y 1 B B *	X Y 1 B B
5) *e*3	X Y 1 B B *	X Y 1 B B
6) *e*6	X Y 1 B B *	fin

Analicemos los resultados obtenidos. En el primer caso la máquina de Turing llegó al estado final *e*5 y, en el segundo, al estado final *e*6. Si ahora se conecta el estado *e*5 con el sí y el estado *e*6 con el no, se puede decir que la máquina *acepta* la primera cadena (0011) y no acepta la segunda (011). El lector puede comprobar que la máquina acepta todas las cadenas de los tipos $0^n\ 1^n$ ($n \geqslant 1$): aquellas formadas de al menos un cero seguido del mismo número de unos, y que no acepta cadenas que no sigan esta regla de formación. Tal vez fuera necesario insertar un estado de trampa (equivalente a *e*6) para que la máquina rechace otras combinaciones. Así, para cualquier pareja estado actual-símbolo actual no definida ya en la parte de estímulos de la tabla, simplemente se asocia una reacción que lleve a la máquina a este nuevo estado final.

Si se considera que existe una infinidad de cadenas formadas por ceros y unos, puede imaginarse la potencia del modelo de reconocimiento recién descrito. En efecto, permite atacar y resolver un problema de tamaño infinito con recursos finitos. No sólo eso, también, como desde un punto de vista formal es posible considerar como un lenguaje a un conjunto de símbolos (en este caso únicamente unos y ceros) y una regla de formación (en este caso, que deben ser *n* ceros seguidos de *n* unos), resulta que la máquina de Turing acepta (o reconoce) a uno de ellos, y puede entonces considerarse como un esquema de manejo de lenguajes formales.

La máquina de Turing representa el concepto de modelo llevado a su expresión más primitiva, pues —definiendo adecuadamente sus elementos, tabla y símbolos—, se puede representar cualquier proceso descriptible.

En este punto cabe volver a preguntar: ¿existirá siempre una máquina de Turing que se detenga (llegue a un estado final) para cualquier proceso adecuadamente descrito y codificado?

Proceso computable

La respuesta sigue siendo no. Ahora que ya es posible definir con más propiedad los conceptos de algoritmo y de computabilidad, se dice que un proceso es computable o tiene solución algorítmica cuando puede ser representado por medio de una máquina de Turing que llega, en algún momento, a un estado final.

Si la máquina de Turing llega a un estado final con un sí, se estará haciendo una correspondencia entre ella y el modelo de decisión. Pero cuando la

máquina no llega a este estado final pueden suceder dos cosas: que llegue a un estado de trampa, de donde ya no salga, o que sencillamente nunca se pueda saber si terminará o no con la computación.

Para el primer caso bastará con hacer una equivalencia entre este estado de trampa y el no del modelo de decisión; pero para el segundo hay que decidir entre seguir esperando o no el resultado. En 1936 Turing demostró matemáticamente que existen procesos para los cuales la máquina *nunca* terminará con un sí y *nunca* terminará con un no. En este caso se podría esperar toda la eternidad para ver si la máquina se detiene o no, sin poder llegar a saber si se detendrá en el siguiente instante. Se dice sencillamente que el problema no es computable, o bien, que no es posible decidir, en un tiempo finito, si el proceso es representable algorítmicamente.

Los problemas de este tipo reciben el nombre de **problemas indecidibles** o "problemas no solucionables en forma algorítmica", y el simple hecho de haberlos descubierto representa una prueba de las enormes capacidades del método matemático para explorar la realidad formal del mundo, ya que se está hablando de verdaderos fantasmas que es posible describir, pero nunca representar por completo para todos los casos.

Sin embargo, Turing no detuvo aquí sus investigaciones, sino que en la búsqueda de los problemas indecidibles generalizó el concepto de máquina, como se verá ahora.

Supóngase que se construye la tabla de una máquina de Turing (que llamaremos MT2) capaz, por ejemplo, de extraer la raíz cuadrada de un número adecuadamente codificado en la cinta, dejando el resultado del proceso en otra sección de la misma. Está claro que esta máquina sólo tiene esa función, y no es capaz de sumar dos números, ni de hacer ninguna otra cosa diferente de la original.

Bien, ahora hagamos un esfuerzo y observemos mentalmente la tabla de MT2; ¿acaso no consiste en una serie de símbolos que conservan cierto orden? ¿Sería posible codificarla usando los símbolos de otro diccionario más complejo? Si se hiciera esto, se podría colocar estos datos (la tabla MT2) codificados en la cinta de otra máquina de Turing. Hagámoslo. Se tendrá así una nueva máquina (que llamaremos MTu), igual a las anteriores, y que tiene codificada en su cinta la tabla de otra máquina de Turing. Ahora construimos la tabla MTu, o sea, la tabla que guíe a esta nueva máquina.

La nueva tabla tendrá instrucciones (parejas estímulo-reacción) que le indiquen cómo seguir las instrucciones de la tabla MT2 codificada. Dichas parejas estímulo-reacción irán haciendo que la máquina MTu *se comporte exactamente de la misma manera que se comportaría MT2*, puesto que es precisamente la tabla MT2 la que está codificada en la cinta. Lo mismo se aplicaría, por supuesto, para cualquier otra máquina cuya tabla estuviera en su lugar en la cinta.

Una máquina universal

¡Se ha logrado una máquina universal!, pues imita o simula el comportamiento de cualquier otra que esté codificada en su cinta. Pensemos un momento en la importancia de lo obtenido: se tiene un *modelo de modelos*.

La máquina universal de Turing (MTu) es, por excelencia, el modelo teórico de la computabilidad; basta con codificar cualquier máquina particular

de Turing en su cinta para que sea entonces simulada y, por ende, pueda resolver ese problema particular algorítmicamente.

Ahora es cuando Turing se hace la siguiente pregunta: ¿podrá la máquina universal determinar si la máquina —particular— que está siendo simulada se va a detener (en un estado final) o no?

Supóngase que sí puede: que existe una MT1 que determina si cualquier otra (que llamamos MT0) se va a detener o no; es decir, termina con un sí si MT0 se detiene, y con un no si MT0 no se detiene. Entonces, también lo podrá hacer para la codificación de sí misma (es decir, podrá determinar si MT1 se detendrá para cualquier caso particular o no).

Ahora se construye una nueva máquina, MT2, que se detiene si MT0 no lo hace, y viceversa. Esto se puede lograr si se hace que MT2 entre en un ciclo infinito cuando MT0 se detiene (mediante un par de estados de trampa, que hacen que la operación de la máquina oscile entre uno y otro).

¿Qué sucede si MT2 trabaja sobre la codificación de sí misma? Pues que se detendrá si MT2 no se detiene, y no se detendrá si, y sólo si, se detiene. O sea, una contradicción total. Lo que quiere decir que tal máquina no puede existir; lo que a su vez equivale a decir que el problema que estamos estudiando es indecidible*.

Un problema indecidible

En resumen, los pasos fueron:

1. Se supone que se puede construir MT1, que determina si MT0 se detiene o no y que, por tanto, también puede determinar si ella misma lo hace, cuando trabaja sobre su propia codificación.
2. Se construye MT2, que termina con sí si MT0 no se detiene, y con no si MT0 se detiene, mediante un par de estados especiales de trampa que causan que entre en un ciclo infinito cuando MT0 llega a un estado final.
3. Cuando MT2 trabaja sobre su propia codificación se llega a una contradicción, lo cual significa que la suposición del punto 1 es inválida.

El trabajo de Alan Turing abrió muchísimas puertas en la investigación de la computabilidad y demostró la existencia de problemas indecidibles y de un modelo genérico de la computación. A tal grado se reconoce la máquina de Turing como modelo de computabilidad que se acepta generalmente —pero es indemostrable— la siguiente propuesta, conocida como "la hipótesis de Turing": si existe una máquina de Turing para representar un problema, entonces éste tiene solución algorítmica, y su dual: si un problema tiene solución algorítmica, es porque existe una máquina de Turing que la representa.

Equivalencia entre algoritmo y máquina de Turing

Esto, en efecto, hace equivalentes las nociones de algoritmo y de máquina de Turing.

Ya formalizado el concepto de algoritmo, éste se ha convertido en fundamental en matemáticas y en filosofía de las ciencias formales, donde se emplea, junto con la máquina de Turing, en demostraciones y planteamientos de teoremas. Se ha dedicado un anexo al final de este capítulo para describir

* Este análisis del llamado "problema del alto de la máquina de Turing" (*the halting problem*) sigue a lo expuesto en [MINM67, págs. 146-149]. El artículo original de Turing se llamó "On computable numbers, with an application to the *Entscheidungsproblem*"; este último término se refiere al problema de la decidibilidad.

la historia de la lógica matemática, junto con algunas consideraciones de carácter más amplio, para aquellos lectores interesados en la filosofía de las matemáticas.

5.3 Lenguajes formales y autómatas

Se describió ya cómo la máquina de Turing puede reconocer cadenas que pertenecen a un lenguaje, y ahora se darán algunas definiciones básicas sobre lenguajes formales y sus reconocedores, que reciben el nombre genérico de **autómatas**.

El estudio formal de los lenguajes se debe, entre otros investigadores, a Noam Chomsky, lingüista que en 1956 publicó un estudio considerado como pionero en lo que de ahí en adelante se conoce como lingüística matemática*. En [CHON57] Chomsky explica que "el objetivo fundamental en el análisis lingüístico de un lenguaje es separar las frases gramaticalmente correctas de las que no lo son, y estudiar la estructura de las correctas", para luego proponer que "la investigación sintáctica de un lenguaje tiene como objetivo la construcción de una gramática, que generará las sentencias del lenguaje". Es decir, una gramática es el mecanismo mediante el cual se producen las frases que constituyen un lenguaje. Desde este punto de vista, un lenguaje formal es un sistema matemático y, como tal, está sujeto a un tratamiento matemático riguroso, que es a lo que se dedica la teoría de lenguajes y autómatas.

Definiciones preliminares

Las definiciones básicas que siguen permitirán hacer un estudio preliminar sobre estos temas, que son centrales para la teoría de la computabilidad y para su aplicación a los lenguajes de programación, los compiladores y, en general, para la programación de sistemas avanzada.

Un **alfabeto** o **vocabulario** es un conjunto finito de símbolos, y una **frase** o **sentencia** es una cadena de longitud finita compuesta de símbolos del alfabeto. Para propósitos de demostración de teoremas y otros estudios, existe una frase especial que consta de cero elementos, llamada **cadena vacía**, que se representará mediante el símbolo Λ.

Si V es un alfabeto, entonces la notación V^* (léase "la cerradura de V") denota al conjunto de todas las frases compuestas de símbolos de V, incluyendo la cadena vacía. En general, la cerradura representa la operación de repetición. Por ejemplo, si V es un alfabeto que contiene dos símbolos, 0 y 1, entonces

$$V = \{0, 1\}$$

* Chomsky es además un importante crítico de la política exterior estadounidense, y representa una corriente de opinión de la izquierda liberal de su país. Ha publicado varios libros sobre política exterior y sobre lo que él llama "la responsabilidad de los intelectuales" en el quehacer político.

y la cerradura de V será un conjunto infinito:

$$V^* = \{\Lambda, 0, 1, 00, 10, 01, 11, 000, 100, 010, 001, 110, 101, 011, 111, 0000 \ldots\} .$$

A un conjunto cualquiera de frases se le conoce como **lenguaje**.

Surgen entonces varias preguntas: ¿Cómo se representan las frases de un lenguaje, y más aun, si éste es infinito (es decir, si contiene un número infinito de frases)? ¿Existe una representación finita para cada lenguaje? ¿Qué se puede decir de la estructura de los lenguajes que sí admiten una representación finita?

Como se verá a continuación, las representaciones finitas de los lenguajes son esquemas *generativos*, es decir, son un modelo dinámico para generar frases de un lenguaje, y se llaman **gramáticas formales**.

Una gramática G se define como un conjunto de cuatro elementos:

- un vocabulario, Vn, llamado *no terminal*.
- un vocabulario, Vt, llamado *terminal*.
- un conjunto P de *reglas de producción* y
- un *símbolo especial de inicio*, S, que por definición pertenece al vocabulario no terminal.

O sea, $G = (Vn, Vt, P, S)$

El corazón de la gramática es el conjunto P de reglas de producción o de reescritura, donde cada regla consta de dos miembros, conectados mediante una flecha:

$$a \rightarrow b$$

El miembro izquierdo está formado por al menos un elemento tomado de los alfabetos terminal y no terminal o, en términos más generales, por alguna cadena no vacía tomada de la cerradura de Vn y Vt. El miembro derecho tiene las mismas características; pero incluye la posibilidad de que pueda ser la cadena vacía.

Empleando notación elemental de conjuntos:

$$a \in \{ Vn \ U \ Vt \}^* - \{ \Lambda \}$$
$$b \in \{ Vn \ U \ Vt \}^*$$

Mediante las reglas de producción de una gramática se pueden obtener cadenas de símbolos a partir del símbolo inicial que, por tanto, debe ser el miembro izquierdo de al menos una regla dentro del conjunto P.

Cómo se obtienen frases con una gramática

El procedimiento mediante el cual se producen cadenas de símbolos (frases de un lenguaje) es el siguiente:

1. Se escoge una regla de producción que contenga a S como miembro izquierdo.
2. Se *aplica* la regla, es decir, se sustituye el símbolo S por la cadena b del lado derecho de esa regla.
3. Ahora pueden suceder dos cosas: que esa cadena b esté formada exclusivamente por elementos del vocabulario terminal Vt, o que no sea así. En el primer caso se ha obtenido ya una frase (que, por tanto, forma parte del lenguaje producido por la gramática) y el procedimiento termina. Es posible volver a aplicarlo, comenzando por el paso 1, para intentar obtener otras frases terminales diferentes. Por otra parte, si la cadena b contiene al menos un elemento no terminal entonces el procedimiento de generación aún no termina, y habrá que escoger alguna regla dentro del conjunto que contenga ese elemento no terminal como parte de su miembro izquierdo, para entonces aplicarla y volver a hacer la consideración del paso 3. Este proceso se repite hasta que se obtenga una cadena compuesta exclusivamente de elementos terminales. Si no se llega a un fin, entonces se dice que la gramática no produce nada.

La razón de ser de una gramática es producir cadenas terminales, que entonces forman parte del lenguaje producido por ella.

Un ejemplo aclarará esto. Considérese la siguiente gramática:

$G = (\{ S \}, \{ 0,1 \}, P, S)$

en donde P consta de las siguientes reglas:

1. $S \rightarrow 0S1$
2. $S \rightarrow 01$

Se puede aplicar inicialmente la regla 1 o bien la 2, porque ambas contienen a S como miembro izquierdo.

Si se aplica la regla 2 se obtiene
$S \Rightarrow 01$ (La doble flecha representa la aplicación de alguna regla.)

y el proceso termina allí, ya que la cadena 01 contiene únicamente elementos terminales.

Si se aplicara inicialmente la regla 1, sin embargo, se obtendría lo siguiente:

$S \Rightarrow 0S1$

y como el lado derecho de esta aplicación contiene al menos un elemento no terminal, se vuelve a aplicar una regla (la 2), para obtener

$0S1 \Rightarrow 0\underline{0}\underline{1}1$ (Lo subrayado es la parte de la cadena que se sustituyó.)

que ya es una cadena terminal y, por tanto, perteneciente al lenguaje que produce esta gramática.

El proceso completo fue, entonces, éste:

$$S \Rightarrow 0S1 \Rightarrow 0011$$

El lector puede comprobar que la cadena 000111 también pertenece al lenguaje producido por esta gramática, porque

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000111$$

pero que la cadena 00111 no puede ser producida por esta gramática, como tampoco alguna que comience con 1.

Ahora es posible intentar contestar las preguntas planteadas al inicio de este apartado, diciendo que no obstante que esta gramática produce un número infinito de frases (porque se puede escoger la regla 1 tantas veces como se desee), todas ellas tienen la misma estructura gramatical (que en lingüística se conoce como *estructura profunda*): un grupo de ceros seguido del mismo número de unos, y que existe un procedimiento para obtenerlas. Como se dijo, el conjunto de todas las frases terminales que se pueden obtener con una gramática es el lenguaje producido por ella, lo cual se representa formalmente así:

$$L(G) = \{ w \mid w \in Vt^* \ \& \ S \Rightarrow^* w \}$$

Definición formal de lenguaje

Y esto se lee como sigue: "el lenguaje L producido por una gramática G es el conjunto de todas las cadenas w tales que w pertenece a la cerradura del vocabulario terminal (o sea, que w consta sólo de terminales) y que w fue además obtenida mediante la aplicación repetida de reglas de producción (lo que se indica mostrando la cerradura de la aplicación de alguna regla), habiendo comenzado con el símbolo inicial". En otras palabras, el lenguaje producido por una gramática es el conjunto de todas las cadenas terminales válidas que se pueden derivar de ella.

En general, averiguar cuál lenguaje produce una gramática no es una tarea trivial, porque hay que determinar formalmente la estructura común a todas sus frases. Existe, por supuesto, toda un álgebra para manejar gramáticas y lenguajes, pero ya no hay espacio aquí para describirla. Las referencias [HOPJ69] y [HOPJ79] del capítulo 4, así como [CODH86] y [DEND78], mencionadas al final de éste, se dedican a estos temas que, en general, son de nivel avanzado.

Jerarquización de gramáticas

Una vez que se cuenta con un procedimiento para producir frases dotadas de una estructura afín (o sea, una gramática), es posible estudiar a fondo sus

propiedades matemáticas, con el fin de intentar determinar de antemano las características que se pueden esperar de los lenguajes que se obtengan. Para este fin, Chomsky clasificó las gramáticas (y, por ende, los lenguajes que generan) en varias familias, enmarcadas en una jerarquía de cuatro tipos. En esta jerarquía, las gramáticas de tipo 0 son las más generales e incluyen como caso particular a las de tipo 1, que a su vez tienen las propiedades de las que engloban —de tipo 2—, para llegar al extremo más limitado, las gramáticas de tipo 3.

Tipos de gramáticas

En una gramática de **tipo 0** no hay restricciones en la forma de definir los miembros izquierdo y derecho de una regla de producción, excepto, claro, que estén formadas por elementos de los vocabularios ya definidos.

Una gramática es de **tipo 1** si se exige que la longitud del miembro derecho de toda regla de producción sea mayor o igual que la longitud del izquierdo. Con esto se impide que las cadenas que se obtengan en el transcurso de la aplicación de las reglas sean de longitud decreciente, y se impide asimismo el caso extremo de que mediante una gramática de tipo 1 puedan desaparecer cadenas de símbolos. Esto tiene implicaciones teóricas importantes. Las gramáticas de tipo 1 también reciben el nombre de "sensibles al contexto", porque del lado izquierdo puede haber más de un elemento, lo que implica que un símbolo puede reemplazarse en el contexto de otros, como se explicará en un ejemplo.

Una gramática es de **tipo 2** cuando se exige que el miembro izquierdo de toda regla de producción sea un único elemento no terminal, a la vez que se impide que el lado derecho esté vacío. Es fácil observar que estas nuevas limitaciones cumplen además la restricción pedida a las gramáticas anteriores: que la longitud del lado derecho sea al menos igual a la del lado izquierdo. Como del lado izquierdo de las reglas en estas gramáticas de tipo 2 sólo puede haber un elemento no terminal, se puede sustituir un solo símbolo no terminal a la vez, independientemente de lo que lo rodea; por esta razón, a estas gramáticas se les conoce también como "independientes del contexto".

Por último, si a una gramática de tipo 2 se le restringe aun más en sus reglas de producción, regularizándolas para que sean de alguna de dos formas bien definidas, entonces se convierte en una gramática de tipo 3, también llamada "gramática regular".

Una gramática es de **tipo 3** cuando las dos formas permitidas para las reglas son: 1) un no terminal del lado izquierdo produce un terminal del lado derecho y, 2) un no terminal del lado izquierdo produce del lado derecho un terminal seguido de un no terminal.

Es decir, cuando todas sus reglas son de alguna de estas dos formas:

$$A \rightarrow a$$
$$A \rightarrow aA$$

para elementos no terminales (A) y terminales (a) cualesquiera.

Antes de mencionar los resultados que se obtuvieron al hacer esta jerarquización se mostrarán algunas gramáticas.

Por ejemplo, la gramática con reglas
1. $S \rightarrow 0S1$
2. $S \rightarrow 01$

mostrada anteriormente es de tipo 2, porque tiene un solo elemento no terminal del lado izquierdo de cada regla y porque sus lados derechos no son nulos.

Esta nueva gramática es de tipo 1, porque la única restricción en las reglas es que el lado derecho sea mayor o igual que el izquierdo:

1. $S \rightarrow Abc$
2. $Ab \rightarrow aAbB$
3. $Bb \rightarrow bB$
4. $Bc \rightarrow bcc$
5. $A \rightarrow a$

(Las mayúsculas son elementos no terminales y las minúsculas son terminales.)

Con ella se pueden obtener cadenas compuestas por las letras a,b,c en ese orden, y con el mismo número de cada una.
Por ejemplo, así se genera la cadena "*aabbcc*":

$$S \Rightarrow Abc \Rightarrow aAbBc \Rightarrow aabBc \Rightarrow aabbcc$$

aplicando sucesivamente las reglas 1, 2, 5 y 4. Obsérvese que cuando se aplicó la regla 2 se usó el contexto formado por la secuencia *Ab*.

Éste es un ejemplo de una gramática regular o de tipo 3, que produce palabras formadas por cualquier número de letras o dígitos, pero que comienzan con una letra:

1. S $\rightarrow l$
2. $S \rightarrow lA$
3. $A \rightarrow l$
4. $A \rightarrow d$
5. $A \rightarrow lA$
6. $A \rightarrow dA$

(La *l* es un terminal que representa una letra cualquiera; la *d* es un terminal que representa un dígito cualquiera, y la *A* es un símbolo no terminal auxiliar.)

Por ejemplo, la secuencia de derivación de la cadena "*alfa27*" es:

$$S \Rightarrow aA \Rightarrow alA \Rightarrow alfA \Rightarrow alfaA \Rightarrow alfa2A \Rightarrow alfa27$$

aplicando las reglas 2, 5, 5, 5, 6 y 4.

Como se dijo, el estudio de las gramáticas y los lenguajes formales es un campo muy amplio de estudio y de investigación, y entre los sorprendentes resultados que se han obtenido resalta el hecho de que las gramáticas de tipo 0 y 1 exhiben un buen número de propiedades indecidibles. Por ejemplo, para una gramática de tipo 0 no se puede determinar algorítmicamente si

Propiedades indecidibles

una frase dada es producto de ella o no; es decir, no se puede saber si una máquina de Turing diseñada para reconocer esas frases se detendrá o no durante el transcurso de esa computación. Del mismo modo, es indecidible el procedimiento para averiguar si dos gramáticas de tipo 0 dadas producen o no el mismo lenguaje. Más aun, el número de propiedades indecidibles de las gramáticas es mayor que el de las que sí lo son; por ello, el hecho de restringirlas mediante la jerarquización de Chomsky es útil, pues produce construcciones más "dóciles" cuyas características sí son determinables algorítmicamente. El precio que se paga, por supuesto, es que una gramática de tipo 3 o de tipo 2 produce un lenguaje más limitado y menos rico que una de tipo 1 ó 0.

Con todo lo analizado hasta ahora debe estar claro que hay dos procesos fundamentales en teoría de gramáticas y lenguajes formales: la *producción* de las frases de un lenguaje a partir de una gramática, y el *reconocimiento* de frases como pertenecientes al lenguaje producido por una gramática. Tal parece que el proceso de producir una frase debe ser el inverso de reconocerla, sobre todo si se visualiza la producción de una frase como la generación de una descripción y su reconocimiento como la representación de ella. De hecho, así es y, como se ha dicho ya, para cada tipo de gramática existe un reconocedor particular que recibe el nombre genérico de autómata. Hay, pues, también una jerarquización de autómatas, que se muestra a continuación.

Grámatica y lenguaje que genera	Reconocedor
Tipo 0	Máquina de Turing
Tipo 1 (Sensible al contexto)	Autómata lineal
Tipo 2 (Independiente del contexto)	Autómata de pila*
Tipo 3 (Regular)	Autómata finito

Queda fuera de los alcances de este curso dar una descripción de estos autómatas, aunque ya se ha dedicado una sección al más poderoso y general de ellos, que los puede simular a todos: la máquina de Turing.

Sin embargo, como se ha visto, aun este último autómata/reconocedor por excelencia tiene limitantes, porque existe toda una clase de procesos que se pueden describir mas no representar algorítmicamente; estos son los procesos indecidibles, que efectivamente marcan una frontera entre lo que es computable y lo que no lo es.

* Recibe este nombre debido a que emplea una estructura de datos conocida como pila (*stack*, en inglés).

La teoría de la computabilidad ha permitido descubrir muchas de las propiedades de los lenguajes y sus reconocedores, y ha permitido también el diseño de los lenguajes de programación y de sus correspondientes autómatas encargados de reconocerlos y traducirlos a expresiones más simples —los compiladores, como se describió en el capítulo 4—. La gramática que describe los lenguajes de programación usuales es de tipo 2, y el analizador sintáctico de un compilador es un autómata de pila. De la misma forma, el analizador lexicográfico de un compilador es un autómata finito, porque la gramática que se emplea en la definición léxica de un lenguaje como FORTRAN o Pascal es de tipo 3 (de hecho, el ejemplo de una gramática de tipo 3 recién expuesto muestra cómo se pueden obtener los nombres simbólicos de las variables para un lenguaje de estos).

En términos más formales, se dice que un lenguaje cuyas frases pueden ser generadas por un procedimiento es **recursivamente enumerable**, mientras que se le llama **recursivo*** si existe un algoritmo para reconocerlo.

Que un lenguaje sea recursivamente enumerable significa que se puede producir mediante una gramática; que sea recursivo significa que existe para él un algoritmo de reconocimiento: una máquina de Turing que llega a un estado final (un sí o un no) cuando analiza las frases que lo componen.

Termina este capítulo introductorio sobre computabilidad enunciando el resultado final (intuitivamente contradictorio, porque la indecidibilidad no parece ser muy natural): no todos los lenguajes recursivamente enumerables son recursivos. En otras palabras, *existen límites inherentes en nuestro mecanismo mental de conocimiento*.

El resto de este libro se dedicará al problema de integrar estos conceptos teóricos a formas prácticas de ser programadas en una computadora porque, sobra decirlo, estos resultados de 1936 son anteriores al nacimiento de las computadoras electrónicas digitales**. A partir del siguiente capítulo se comenzará a formar un conjunto de métodos para escribir algoritmos, que cumplan además ciertos criterios humanos, y no sólo formales.

5.4 Anexo: visión histórica de la lógica matemática

Todas las proposiciones de la lógica dicen lo mismo. Es decir, nada.

Ludwig Wittgenstein

Lógica tradicional

El arte de la lógica nace en la Grecia de los siglos IV y V a.C. como un desarrollo de los esfuerzos de grupos de filósofos y retóricos que pugnaban por

* El empleo del término *recursivo* en este contexto es independiente del que se le da al referirse a lenguajes de programación; es importante no confundir estos dos usos. El primero, el que aquí se emplea, toma su nombre de la teoría de funciones recursivas de las matemáticas, mientras que el segundo, el más común, se refiere a la capacidad que ofrece un lenguaje de programación de manejar módulos y rutinas que se llaman a sí mismos.

** Como dato interesante cabe mencionar que Turing se dedicó, años después de estas investigaciones, al desarrollo de las primeras computadoras electrónicas.

refutarse mutuamente a través de discursos cada vez más elaborados. Buscaban en particular la manera de encontrar las fallas en los razonamientos de sus adversarios. Uno de los métodos usados era el de aceptar el punto de vista de sus contrarios y luego demostrar que implicaba consecuencias absurdas.

Primero Sócrates (470-399 a.C.), y luego su discípulo Platón (428-347 a.C.), comienzan el estudio de estos problemas con un método definido que consistía en buscar las características generales implícitas en los razonamientos.

Platón, en el diálogo "Eutifrón o de la santidad", pregunta "¿Lo santo es amado por los dioses porque es santo, o es santo porque es amado por ellos?", dando a entender que Sócrates conocía perfectamente bien la diferencia entre una condición suficiente y una condición necesaria ([PLAT73]).

Pero no será sino hasta Aristóteles (384-322 a.C.) cuando la lógica nazca como un método especializado que servirá como instrumento de la filosofía. Aristóteles es creador de un sistema lógico llamado silogística, y hace también un estudio del concepto de definición. Se da cuenta de que la definición debe estar dada en términos anteriores al objeto que se define, observando así la necesidad de partir de algunos términos indefinidos para construir la serie de definiciones. Hace notar que una definición dice qué es una cosa, pero no que esa cosa existe, por lo que se vuelve necesario, además, demostrar su existencia.

Con respecto a la lógica, Aristóteles da a entender que se deriva de las matemáticas. Sus principios básicos son la ley de contradicción (una proposición no puede ser falsa y verdadera a la vez) y la ley del tercero excluido (una proposición debe ser o bien verdadera o bien falsa). Con base en esos principios es posible construir un método de pruebas e inferencias deductivas. También introduce el uso de variables en el estudio de la lógica.

Aristóteles usa la lógica como un instrumento (*organon*) y le da el nombre de *analítica*. Su estudio se divide en varias partes:

- Categorías (estudio de las condiciones generales del pensamiento)
- Analíticos primeros (estudio del silogismo)
- Analíticos segundos (estudio de la demostración)
- Tópicos y refutaciones de los sofistas (en donde se expone un método de argumentación)

Hay que mencionar que la lógica de Aristóteles básicamente no contiene errores y ha sido reaxiomatizada para mostrar que es consistente y decidible*.

Después de Aristóteles

En principio, el sistema de Aristóteles se mantuvo inconmovible por más de 2000 años, hasta el nacimiento de la lógica moderna en el siglo XIX.

La lógica fundada por Aristóteles es una lógica de términos basada en la silogística. La lógica moderna es una lógica de proposiciones que presupone a la escolástica o aristotélica.

* El lógico polaco Jan Lukasiewicz publicó en 1951 el libro *Aristotle's Syllogistic from the Standpoint of Modern Formal Logic* (Oxford University Press, Nueva York), en el que se exponen esos resultados.

.s precursores de la lógica de proposiciones se pueden encontrar poco después de la muerte del maestro (322 a.C.). Para los filósofos Teofrasto de Ereso y Crisipo de Soles, posteriores a Aristóteles, la lógica tiene dos valores: una proposición cualquiera o es verdadera o es falsa. Y ésta es la definición que dan de proposición. Reconocían proposiciones simples y compuestas. Las segundas resultan de combinaciones de proposiciones diferentes, o de dos ocurrencias de la misma; esta combinación se logra por medio de conectivas. También sabían negar proposiciones, anteponiéndoles una partícula negativa; y formaban una función de verdad con el resultado, diciendo que no-A es verdadero cuando A es falso, y viceversa. Tenían, además, conocimientos acerca de la implicación, la conjunción y la disyunción exclusiva.

Mucho tiempo después se encontrará la lógica aristotélica dentro de la cultura árabe, entre los siglos VIII y XIII, y en la lógica de la Edad Media.

Es interesante abrir un paréntesis para considerar la lógica hindú. En la India florecieron varias escuelas lógicas enmarcadas dentro de la filosofía hindú. Su estudio puede dividirse en cinco partes:

- Gramática (estudio de las reglas lógicas de formación en sánscrito)
- Mimamsa (dedicada a problemas de interpretación de textos)
- Vaisésika y antigua Nyaya (sistemas de filosofía natural)
- Lógica budista (se encuentran aquí algunos fundamentos de lógica formal; esta escuela comienza a excluir consideraciones ontológicas y psicológicas del estudio de la lógica)
- Nueva Nyaya (fase final de la lógica hindú, que continúa hasta la fecha)

Dentro de la gramática se puede encontrar una preocupación por un criterio de economía, que requiere la eliminación de elementos superfluos. Se construye así un lenguaje técnico, con abreviaciones y conceptos elementales, que se usa de acuerdo con ciertas reglas de composición y sustitución.

Precursores de la lógica moderna

Aunque la lógica matemática moderna nace en 1847 con la publicación de los trabajos de los matemáticos ingleses Augustus De Morgan (1806-1871) y George Boole (1815-1864), Leibnitz, Euler, Lambert y Bolzano realizaron los estudios que prepararon el camino*.

* Gottfried Wilhelm Leibnitz (1646-1716), filósofo y matemático alemán, fue fundador, junto con Newton, del cálculo infinitesimal. Representante del racionalismo, para Leibnitz el criterio de verdad del conocimiento no consiste en su adecuación a la realidad, sino en su intrínseco carácter necesario. Con fundamento en el principio de razón suficiente, y en el de identidad, formula la noción de la *mónada*, sustancia de la cual afirma está formado todo lo existente. Es creador de un sistema lógico dedicado a la mecanización del pensamiento y a la creación de un lenguaje universal.

Leonhard Euler (1707-1783), matemático suizo extraordinariamente fértil en descubrimientos y estudios. Alumno de Jean Bernoulli, Euler inicia la época del análisis matemático y de la geometría analítica en tres dimensiones.

Johann Heinrich Lambert (1728-1777), físico, matemático y astrónomo alemán. Elaboró una teoría del conocimiento, con influencia de la silogística, dividida en cuatro partes: sobre las leyes formales del pensamiento; sobre los elementos simples del conocimiento; sobre la distinción entre lo verdadero y lo falso; sobre las causas del error.

Bernhard Bolzano (1781-1848), filósofo y matemático austriaco. Consideraba a las proposiciones como poseedoras en sí de una realidad lógica desvinculada del ser pensante.

Leibnitz suponía que un concepto se puede descomponer en elementos simples, tal como un entero es separable en sus factores primos, y sugería la posibilidad de formar un lenguaje universal, estructurado lógicamente y capaz de servir como el instrumento de análisis científico. Tenía todo un programa de distinciones y funciones de los diversos elementos del lenguaje y sentó las bases del concepto de función proposicional. Tenía la intención de fundar un cálculo lógico, capaz de expresar ideas de una manera lógica y sin ambigüedades. También hace una distinción fundamental entre antecedente y sujeto, y entre consecuente y predicado, pasando así al moderno punto de vista de proposiciones, en lugar de usarlos como términos en el sentido aristotélico.

Euler contribuye a la lógica con sus famosos diagramas, que eran ilustraciones geométricas de los silogismos; aunque estrictamente hablando no fueron de su invención. él fue quien les dio un uso generalizado.

Lambert dedicó algunos ensayos a la empresa de crear un cálculo de la lógica y tomó un punto de vista diferente del de Aristóteles para tratar problemas relacionados con la lógica. Utilizó las operaciones de multiplicación, división, suma y resta dentro de la lógica, aunque no tuvo mucho éxito en sus logros. Años después, Boole repetiría exitosamente algunos de estos intentos.

Bolzano consideraba la lógica como una teoría acerca de la ciencia. Utilizó un lenguaje semiformal, de su invención, para sus investigaciones sobre lógica que, desde su punto de vista, trabaja con términos y proposiciones, que constituyen sus entidades fundamentales.

Una de sus ideas era asignar valores de validez a proposiciones que se obtienen de sustituir términos distintos en una proposición dada, para llegar así a una especie de tabla de verdad que decía si la proposición era universalmente válida, universalmente contraválida o consistente.

Lógica del periodo de Boole

Los sistemas de lógica moderna son producto de la segunda mitad del siglo XIX, fuera ya de la influencia directa de Aristóteles. Aquí se encuentran varios autores importantes.

De Morgan, quien además desarrolló trabajos de álgebra y se interesó por el aspecto educativo y pedagógico de las matemáticas, logró estudiar los silogismos en su forma más general, como series de combinaciones de relaciones, hasta mostrar lo que llamaba la "forma pura" de un juicio.

En 1847 apareció el libro de George Boole, *Mathematical Analysis of Logic*, destinado a abrir una nueva época en el conocimiento y estudio de la lógica. En esta obra, y en la posterior y definitiva, *An Investigation on the Laws of thought on which are founded the Mathematical Theories of Logic and Probabilities*, probaba que las leyes del álgebra pueden ser expresadas formalmente sin necesidad de alguna interpretación particular. En ambos estudios representa lo que llama "operaciones necesarias para el pensamiento" por medio de un álgebra simbólica. Boole usa las letras x y y como "símbolos electivos" para

clases formadas y seleccionadas de un universo de cosas. Utiliza el símbolo 1 para denotar el universo o universo del discurso.

Toma de De Morgan el recurso de denotar la clase complementaria como $1-x$. Muestra también el uso y las limitaciones de las operaciones de suma y multiplicación en el álgebra de clases y sus propiedades distributiva y conmutativa.

El mismo Boole se da cuenta de que su sistema no se restringe a una interpretación de clases, sino que también es válido para una interpretación más limitada de los valores de las variables; por ejemplo, 0 y 1. Otra interpretación revela por vez primera la afinidad entre la lógica de clases y el cálculo de proposiciones.

Será tarea de sus sucesores depurar el sistema para extraer algunos elementos extraños (como las operaciones algebraicas de división y sustracción) que había introducido, pero el sistema de la lógica moderna ya estaba creado.

El inglés William Stanley Jevons (1835-1882) es de los primeros en acometer la tarea de revisar el álgebra recién creada y hace ciertos cambios y mejoras importantes que hoy día siguen inalterados. Crea un sistema (ya abandonado) de simplificación de expresiones mediante combinaciones de variables y de búsqueda de inconsistencias entre ellas. Inventa un ábaco lógico para resolver estos problemas; ésta fue una de las primeras "máquinas" que trabajan con variables lógicas.

El matemático inglés John Venn (1834-1923) hizo intentos por encontrar significación a las operaciones algebraicas de división y sustracción en el sistema de Boole, consiguiendo un mayor conocimiento de la teoría, no una utilización práctica de ellas. Es creador de los diagramas que llevan su nombre y son representaciones gráficas de los procesos algebraicos introducidos por Boole e ilustrados mecánicamente en el alfabeto de Jevons. Logra así obtener los resultados que buscaba este último al hacer sus combinaciones de variables, pero de manera más sencilla y clara.

Lewis Carroll (1832-1898), cuyo verdadero nombre era Charles L. Dogson, escribió *Alicia en el país de las maravillas* y *A través del espejo*; matemático inglés, fue autor de la importante observación acerca de la inferencia que dice: una ley que permite obtener conclusiones de las premisas no puede ser tratada a su vez como premisa, pues se generaría una regresión infinita. Recientemente se descubrió un libro suyo que lo hace aparecer como un importante autor; hay una referencia a este libro en el artículo "Lewis Carroll's Lost Book on Logic" de W. W. Bartley, publicado en la revista *Scientific American*, julio de 1972.

Lógica moderna

Gottlob Frege (1848-1925), lógico y matemático alemán, uno de los más importantes autores de la fundamentación de la aritmética, hizo el análisis lógico de la demostración por inducción matemática. Muchos de sus conceptos y símbolos serán compartidos por Russell.

Alfred North Whitehead (1861-1947) y Bertrand Russell (1861-1969), filósofos y matemáticos británicos, escriben entre 1910 y 1913 una de las obras máximas de la era moderna, *Principia Mathematica*, donde tratan de demostrar que las matemáticas están fundamentadas en la lógica y, a la vez, sistematizan todo el conocimiento de lógica hasta esa fecha. En esta obra se desarrolla la lógica partiendo de algunos axiomas y de algunos términos indefinidos, como proposición elemental, función proposicional, negación de una proposición, etc. La obra está contenida en tres volúmenes, y el esfuerzo de crearla les ocasionó, en palabras del mismo Russell, "daño cerebral irreversible".

Sin embargo, la posición logicista de la obra ha sido criticada, además de que el sistema nunca fue completado y tiene partes oscuras pero, sin lugar a dudas, representa un enorme avance en la ciencia de la lógica y el conocimiento.

Como una propuesta diferente del logicismo se encuentra la escuela formalista, de la cual el matemático alemán David Hilbert (1862-1943) fue el exponente más importante. Este autor es el principal axiomatizador de la geometría euclidiana y precursor, por tanto, de las geometrías no euclidianas.

Parte fundamental de su tesis fue la creación de una teoría de la prueba, conocida también como *metamatemática*, cuyo estudio profundo ha llevado a descubrir la estructura interna y las limitaciones inherentes al sistema axiomático, del que se hablará más adelante.

Después de Hilbert, en un sistema axiomático formal los elementos primitivos (símbolos, términos) carecen de contenido esencial explícito y sólo se usan como piezas en un juego totalmente definido por sus axiomas y sus reglas de inferencia.

Hilbert crea un "programa" que se ha hecho famoso en la historia de las matemáticas. El objetivo era acabar definitivamente con los problemas relativos a los fundamentos de esta ciencia; estaba dividido en tres etapas: determinación de los símbolos por emplear; formación de sucesiones finitas de ellos, que serán las expresiones formales; formación de sucesiones finitas de expresiones formales. Se crean así *fórmulas* (sucesiones de símbolos que guardan ciertas reglas) y **teoremas** del sistema, que permiten obtener sucesiones de fórmulas de modo que cada una sea o bien una inferencia lógica de fórmulas precedentes o bien un axioma. Esas sucesiones de fórmulas constituirán así la **demostración** formal de la última fórmula de la sucesión completa.

Esta elaboración de un sistema formal ya estaba incluida en los *Principia*, pero la diferencia reside en que en este caso se trata de un sistema axiomático existencial o formal, mientras que en el anterior el sistema es material, en el sentido de que acude a ciertos elementos lógicos como base de sustentación. Otros autores de la corriente formalista de las matemáticas son Ackermann, Gentzen y von Neumann.

Es importante mencionar también la escuela intuicionista, cuyo principal exponente fue el matemático holandés Jan Brouwer (1881-1966). Aquí se afirma que las matemáticas son el estudio de cierto tipo de construcciones mentales, y la intuición de lo que esa construcción representa no se puede reducir a otros conceptos más primitivos. El intuicionismo no considera que

la lógica sea fundamental para la creación de un sistema de las matemáticas, y hace algunas distinciones entre la forma de utilizar sus principios, que llevan al rechazo del principio del tercero excluido al tratar conjuntos con un número infinito de elementos, pues se podría entonces llegar a considerar proposiciones posiblemente indecidibles. Para el intuicionismo, en fin, las ideas matemáticas son independientes de la vestidura que les pueda ofrecer el lenguaje o la lógica, puesto que son más ricas que esos elementos que les son externos. Entre los autores de esta esta escuela destacan, además, Kronecker, Poincaré y Weyl.

Arend Heyting formuló en 1930 un sistema intuicionista formalizado, y hay otros autores de esta corriente que incluso llegan a no admitir la negación, por ejemplo.

En una de sus conferencias, Hilbert tachó al intuicionismo como "traición a la ciencia", pero debe reconocerse que, aun no siendo muy popular, el cálculo intuicionista es consistente y, en cierto sentido, toda la matemática moderna comparte (a sabiendas o no) algunos de los conceptos de esta escuela. Los libros [BOUN72], [DOUA70] y [KLIM72] son compendios muy amenos sobre filosofía de las matemáticas.

Al matemático polaco nacionalizado estadounidense Emil Post (1897-1954) se debe, además de estudios centrales sobre teoría de la computabilidad, la creación de una técnica de tablas de verdad empleada para el análisis de las definiciones de consistencia y completitud del cálculo proposicional.

Otro autor que descubrió paralelamente el método de las tablas de verdad fue el filósofo y lógico austriaco Ludwig Wittgenstein (1889-1951), considerado como uno de los pensadores más importantes del siglo XX.

En su obra central, *Tractatus Logico-Philosophicus**, reduce la lógica de predicados y las matemáticas a un sistema de cálculo proposicional. Su preocupación acerca de un lenguaje ideal lo lleva luego a realizar estudios de lógica y matemáticas considerándolas fenómenos naturales del lenguaje (durante lo que se conoce como su "segunda etapa"). Parte de los principios del simbolismo y de las relaciones necesarias entre las palabras y las cosas en cualquier lenguaje para mostrar cómo la lógica y filosofía tradicionales cometen equívocos por el deficiente uso del lenguaje. En la primera parte de su estudio comienza la construcción de un lenguaje lógico a partir de lo que llama el "hecho atómico", aquel que no tiene partes que sean hechos. Una proposición será, entonces, una función de verdad de proposiciones atómicas. Este constructivismo que parte de los hechos atómicos coloca a Wittgenstein cerca de las posiciones intuicionistas en la filosofía de las matemáticas.

Rudolph Carnap (1891-1970), filósofo y lógico alemán del llamado Círculo de Viena (formado en torno a la filosofía de Wittgenstein), extendió las técnicas de la lógica moderna a la epistemología, la física y la filosofía de la ciencia.

Alonzo Church (n. 1903), estadounidense, autor de un importante texto de lógica, fue un investigador en problemas de la computabilidad y creador de un esquema formal equivalente al de la máquina de Turing.

---

* Traducida al español (con el mismo título) por Alianza Editorial, Col. Alianza Universidad, núm. 50, Madrid, 1973.

Es importante mencionar la escuela de lógicos polacos, entre los que destacan Alfred Tarski y Jan Lukasiewicz.

Existen muchos otros autores que han contribuido a hacer de la lógica una ciencia viva y abierta a la investigación, y se pueden mencionar los nombres de Frank P. Ramsey, revisor de los *Principia*, Leopold Lowenheim y Thoralf Skolem, autores en el cálculo de predicados, Gerhard Gentzen, creador de una formalización conocida como sistema-L, Willard van Orman Quine, conocido por un método de minimización de funciones lógicas, y H. M. Sheffer, creador de una conectiva del cálculo que lleva su nombre.

La figura más importante de la lógica matemática moderna es, sin duda, el matemático austriaco, nacionalizado estadounidense, Kurt Gödel (1906-1978), cuyos trabajos sobre el teorema de incompletitud y el teorema de la imposibilidad de formalización de una prueba de consistencia para un sistema de axiomas como el que constituye la aritmética, pusieron límites a los métodos axiomáticos; límites que muy pocos matemáticos sospechaban que pudieran existir. Su principal teorema, que lleva el título "Sobre sentencias formalmente indecidibles de *Principia Mathematica* y sistemas afines", de 1931, (conocido simplemente como el teorema de Gödel) prueba que los sistemas formales de las matemáticas son incompletos, puesto que puede encontrarse en ellos sentencias indecidibles que no es posible deducir de los axiomas (ni ellas ni sus negaciones); además, Gödel demostró que no puede probarse la consistencia de un sistema formal, y esto echó por tierra el programa de Hilbert, al mostrar que es imposible construir sistemas formales completos y consistentes. En la referencia [HEMN69] se incluye una explicación de las principales características de la demostración, y en [GODK81] aparece el teorema completo.

Así, la lógica matemática puede dividirse en dos etapas irreconciliables: antes de Gödel (cuando se esperaba algún día culminar con el programa de Hilbert) y después de su trabajo, donde ya se sabe que esto no puede ser.

Estos resultados de 1931 se verían correspondidos con los obtenidos por Turing cinco años después, al mostrar que también existen límites inherentes a la computabilidad y, por tanto, a nuestra capacidad (hasta entonces considerada ilimitada) de análisis matemático y formal.

Algunas definiciones

La lógica de predicados trata de expresiones, su validez y su forma. Al estudiar la estructura interna de las expresiones tiene que vérselas con los cuantificadores y con las conectivas sentenciales. Trata, entonces, del análisis de expresiones simples y de la teoría de las propiedades lógicas de los cuantificadores. Tiene una porción elemental, llamada lógica de predicados de primer orden; ésta es la lógica elemental moderna. Se llama "de primer orden" porque los cuantificadores se aplican solamente a elementos individuales, y no a clases de elementos. Dentro de la lógica de predicados se encuentra la llamada lógica sentencial, que trata de expresiones simples y de los compuestos que se pueden hacer con ellas. Es en esta parte de la lógica donde se emplean

las tablas de verdad y las funciones de verdad. Se entiende la noción de verdad lógica en el estrecho sentido en que lo permiten las tautologías (proposiciones que siempre son ciertas) dentro del sistema de tablas de verdad. Como existe un método algorítmico para determinar las tautologías (por medio de las tablas de verdad), entonces la lógica sentencial es decidible. Un argumento en lógica sentencial es válido si la condicional que corresponde a ese argumento es una tautología.

Sin embargo, la lógica elemental generalizada (que toma en cuenta también a los cuantificadores y sus conectivas) es indecidible: no existe un procedimiento de decisión para la validez de los argumentos en la lógica de primer orden. Con las limitaciones propias a su indecidibilidad, un argumento es válido si su correspondiente condicional es un teorema del cálculo de primer orden; o sea, si es una fórmula bien formada, válida en la lógica de primer orden.

Vienen después las nociones modernas de sintaxis y semántica y sus derivados, como consistencia, interpretación, satisfacibilidad, consecuencia, validez, etc. La sintaxis considera ciertas relaciones entre símbolos, sin acudir a sus significados. La semántica considera las relaciones entre los símbolos y los objetos a los cuales se aplican.

Describir la sintaxis de un lenguaje formal L es especificar sus símbolos y reglas de formación.

Describir sintácticamente un sistema formal S es especificar, además, su aparato deductivo.

Dotar a L de una semántica es darle una interpretación que asigna significados a sus símbolos y expresiones.

Si para una interpretación dada I del lenguaje L los axiomas de un sistema formal S (en términos de L) son válidos, entonces se dice que la interpretación I constituye un **modelo** para S.

Todo esto es la base para la llamada *metalógica*, que responde en parte a una antigua pregunta de Platón sobre la existencia de una ciencia de la ciencia (diálogo "Carmides o de la templanza" [PLAT73]).

En general, se dice que una teoría formal es decidible si existe un algoritmo que permita averiguar si una fórmula cualquiera de la teoría es formalmente demostrable o no. Los axiomas de un sistema formalizado se llaman **completos** si todo enunciado verdadero expresable en el sistema es formalmente derivable de ese conjunto de axiomas. Por último, un sistema axiomático se llama **consistente** si no implica enunciados contradictorios.

Dentro de la lógica llamada no elemental se encuentran los cálculos de predicados de segundo orden en adelante, las teorías de clases, la teoría de funciones recursivas, y la teoría de modelos. Los cálculos de orden superior al primero son necesarios para cubrir los campos de cuantificación sobre clases y sobre clases de clases. Se construyen añadiendo al lenguaje formal de la lógica de primer orden notaciones para distinguir entre tipos de variables y órdenes de estos tipos y, además, variando las leyes de formación y el aparato deductivo del sistema.

También hay estudios sobre lógicas multivalentes; aquellas donde los valores de verdad de las variables lógicas no se restringen a dos (verdadero, fal-

so), sino que se permiten valores adicionales. Estos tipos de lógica cumplen objetivos muy específicos, y algunas son desarrolladas sobre pedido; éste es el caso de una lógica polivalente empleada en la física cuántica, donde son necesarios varios niveles de verdad, correspondientes a algunos grados de libertad particulares en el estudio especializado del campo subatómico.

Visión de un libro fundamental, la obra de George Boole

Como información histórica resulta interesante presentar un resumen del libro de Boole que lleva el enorme título de *Una investigación de las leyes del pensamiento en que están fundamentadas las teorías matemáticas de la lógica y de las probabilidades*, escrito en 1854. Las referencias son de la edición facsimilar que publicó la editorial Dover, Nueva York, 1958.

Comienza por dar una definición de lo que hay que entender por signo:

Un signo es una marca arbitraria, con una interpretación fija, y es susceptible de combinarse con otros signos, sujetándose a ciertas leyes fijas supeditadas a sus mutuas interpretaciones.

Y luego lo explica de esta manera:

Proposición I: Todas las operaciones del lenguaje como instrumento del razonamiento pueden ser dirigidas por un sistema de signos compuesto por los siguientes elementos:

1. *Símbolos literales, como* x, y *y* c, *que representan cosas sujetas a nuestras concepciones.*
2. *Signos de operación, como* $+$, $-$, $\times$, *para aquellas operaciones de la mente por medio de las cuales las concepciones de las cosas se combinan para formar nuevos conceptos implicando los mismos elementos.*
3. *El signo de identidad,* $=$.

Estos símbolos de lógica se usan sujetos a leyes definitivas, parcialmente concordando y parcialmente difiriendo de las leyes para los símbolos correspondientes en la ciencia del álgebra.

La ley expresada por xy = yx *puede caracterizarse diciendo que los símbolos literales* x, y, z, *son conmutativos, como los símbolos del álgebra. Y como la combinación de dos símbolos literales en la forma* xy *expresa la totalidad de esa clase de objetos para los cuales los nombres o cualidades representados por* x *y* y *son aplicables conjuntamente, se sigue que si los dos símbolos tienen exactamente la misma significación, su combinación no expresa más que lo que cualquiera de los dos tomados por separado diría. En tal caso deberíamos entonces tener*

$$xy = x$$

pero como se supone que y *tiene el mismo significado de* x, *podemos reemplazarla en la ecuación por* x, *y tener*

$$xx = x$$

Ahora bien, en el álgebra común, la combinación xx *se representa más brevemente como* x^2.

De acuerdo con esta notación, entonces, la anterior ecuación adquiere la forma

$$x^2 = x$$

que es, en realidad, la expresión de una segunda ley general acerca de aquellos símbolos por medio de los cuales representamos simbólicamente nombres, cualidades o descripciones. (Pág. 31)

Signos de aquellas operaciones mentales por medio de las cuales reunimos partes en un todo, o separamos un todo en sus partes. [...] Estrictamente, las palabras y *y* o, *interpuestas entre los términos descriptivos de dos o más clases de objetos, implican que esas clases son diferentes, de tal manera que ningún miembro de una se encuentra en la otra. En éste y en todos los respectos, las palabras* y *y* o *son análogas al signo + en álgebra; y sus leyes son idénticas. [...] Pero no podemos concebir posible recolectar partes en un todo, sin concebir como posible también el separar las partes del todo. Esta operación se expresa en lenguaje común con el signo* excepto. *Como hemos expresado la operación de agregar con el símbolo +, así también podremos expresar la operación antes descrita con el símbolo −.* (Pág. 33)

Más adelante dice

Proposición II: Determinar el valor lógico y la significación de los símbolos 0 y 1.

El símbolo 0, *tal como se usa en álgebra, satisface la siguiente regla formal*

$$0 \cdot y = 0, \text{ o bien } 0y = 0,$$

para cualquier número que y *pueda representar. Esta ley formal puede ser obedecida en el sistema de la lógica; debemos asignar al símbolo* 0 *una interpretación tal que la representada por* 0y *pueda ser idéntica con la clase representada por* 0, *sin importar cuál sea la clase* y. *Una breve consideración mostrará que esta condición se satisface si el símbolo* 0 *representa* nada. *De acuerdo con una definición anterior, podemos hacer de* nada *una clase. De hecho,* nada *y* universo *son los dos límites de la extensión de clases, puesto que son los límites de las posibles interpretaciones de los nombres en general, ninguno de los cuales puede relacionar menos individuos de los que incluye* nada, *o más de los que incluye* universo.

Proposición III: Si x *representa cualquier clase de objetos, entonces 1-*x *representará la clase contraria o suplementaria de objetos. Por ejemplo, la clase que incluye aquellos objetos que no están comprendidos dentro de la clase* x. (Págs. 47-48).

Inmediatamente después analiza el principio de contradicción, que afirma que es imposible que algún ente posea una cualidad, y que al mismo tiempo no la posea; e indica que esto es una consecuencia de aquella ley fundamental del pensamiento cuya expresión es $x^2 = x$.

Posteriormente pasa al análisis de las proposiciones y su división de acuerdo con los cuantificadores. Las separa en dos grandes grupos: primarias o concretas y secundarias o abstractas.

Luego, en el siguiente capítulo, trata las funciones lógicas del tipo $f(x)$, donde x es un símbolo en el sentido previamente definido, y en los siguientes capítulos habla de la eliminación, y de la reducción y de métodos para abreviar funciones, de sus aplicaciones, etc. La base ha sido puesta ya y la lógica entra en una etapa nueva, luego de más de 2000 años de existencia.

Palabras y conceptos clave

En esta sección se agrupan las palabras y conceptos de importancia que se estudiaron en el capítulo, con el objetivo de que el lector pueda hacer una autoevaluación que consiste en ser capaz de describir con cierta precisión lo que cada término significa, y no sentirse satisfecho hasta haberlo logrado. Se muestran en el orden en que se describieron o se mencionaron.

COMPUTABILIDAD
MODELO
ALGORITMO
MÁQUINA DE TURING
ESTADOS
PROBLEMA INDECIDIBLE
MÁQUINA UNIVERSAL
HALTING PROBLEM
LENGUAJE FORMAL
ALFABETO
FRASE
CADENA VACÍA
GRAMÁTICA FORMAL
ESTRUCTURA DE UN LENGUAJE
JERARQUÍA DE CHOMSKY
GRAMÁTICA REGULAR
GRAMÁTICA SENSIBLE AL CONTEXTO
GRAMÁTICA INDEPENDIENTE DEL CONTEXTO
AUTÓMATA/RECONOCEDOR

Ejercicios

1. Diseñe una máquina de Turing para sumar dos números enteros. La estrategia más sencilla consiste en representar los números en notación unaria y considerar el proceso de sumarlos como se ve en el siguiente ejemplo con los números 3 y 4.

Cinta original

...	X	X	X	+	X	X	X	X	B	...

Cinta después del proceso

...	X	X	X	X	X	X	X	B	B	B	...

(El símbolo + sirve como separador entre los dos números, y el símbolo *B* representa celdas en blanco.)

Observe que la definición de suma que se extrae del ejemplo es bastante sencilla, no hace referencia a consideraciones matemáticas ni teóricas y que, además, no existe impedimento alguno para que así sea. Los conceptos de número y de suma son muy simples, como se muestra en el artículo "The Origins of Number Concepts", de Charles J. Brainerd, publicado en la revista *Scientific American*, marzo de 1973.

2. Turing propuso una "prueba" para determinar si una computadora es inteligente o no, que consistitía en establecer un diálogo, empleando un teclado, con un interlocutor invisible que no se sabe si es una máquina o una persona. Si no es posible averiguar por este procedimiento si el que contesta es uno de nosotros, entonces, allí hay inteligencia. Su definición de esta prueba se puede encontrar en la referencia [PYLZ75], citada en el capítulo 1.

   Organice una discusión documentada sobre esto. Considere, entre otros, el programa ELIZA, de 1964, que simula una sesión psicoanalítica y que ha logrado engañar a muchos, aunque su autor se sorprende por ello, ya que el programa es muy primitivo y esquemático; Existe mucho material sobre ELIZA (en la revista *Scientific American*, en [PYLZ75] y en casi cualquier otro lugar donde se hable de los inicios de la inteligencia artificial); se recomienda el libro de su creador, Joseph Weizenbaum, *Computer Power and Human Reason*, W.H. Freeman, San Francisco, 1976.

3. Diseñe una gramática que produzca palindromas binarios. (Un palindroma binario es una cadena que se lee igual de izquierda a derecha que al revés y que está formada por símbolos de dos tipos únicamente; por ejemplo, 010, 0101010 y 00000100000 son palindromas binarios, así como 0, 1, 11, 00, etc., que son casos especiales.)

4. La demostración de que una gramática G produce un lenguaje L tiene dos partes. En la primera se muestra que esa gramática sólo puede producir cadenas con la estructura del lenguaje L que se propone, y en la segunda se debe demostrar que, dada una cadena perteneciente al lenguaje L, ésta debe ser producto de la gramática G. Es decir, hay que demostrar que la gramática produce ese lenguaje y demostrar también que el lenguaje debe ser producto de esa gramática, o sea, que están hechos el uno para el otro. Demuestre que la gramática ya mencionada $G = (\{S\}, \{0,1\}, P, S)$, donde P consta de las reglas

   1. $S \rightarrow 0S1$
   2. $S \rightarrow 01$

   produce el lenguaje $0^n\,1^n$ ($n \geqslant 1$), es decir, cadenas formadas de un grupo de ceros seguido del mismo número de unos, y nada más.

5. Demuestre que la gramática obtenida en el problema 3 produce todos los palindromas binarios, y nada más.

6. ¿De qué tipo, dentro de la jerarquización de Chomsky, será la gramática de los llamados lenguajes naturales (español, inglés, etc.)? ¿Por qué?

7. El problema de la traducción automática de un idioma a otro ha sido siempre de mucho interés práctico, porque sería muy deseable disponer de un

programa de computadora que recibiera como entrada textos en un idioma y produjera como resultado ese mismo texto, ya traducido. Sin embargo, esto no ha sido posible, y existen razones poderosas, no sólo técnicas, para impedirlo, al menos si se habla de traducciones completas y de calidad.

Haga una investigación sobre el tema y considere, por un lado, la referencia [GRAF72], que habla acerca de la imposibilidad de lograr traductores completos y, por otro, los sistemas profesionales de traducción que se ofrecen a la venta para computadoras grandes (por ejemplo, los de la universidad de Utah). ¿No es esto una contradicción?

8. (Para lectores interesados en filosofía) El hecho de que existan limitaciones en las capacidades de la máquina universal de Turing tiene implicaciones profundas, que van más allá de las estrictamente computacionales, y que entran de lleno en el terreno de la filosofía de las matemáticas y de la ciencia en general. En 1931 Gödel publicó su famoso teorema, que demuestra la imposibilidad de conseguir un cálculo suficientemente rico en el que todos los enunciados y teoremas sean decidibles dentro del sistema, lo que contribuyó al derrumbe del concepto de los esquemas axiomáticos como pilares sobre los que pueden construirse sistemas de conocimiento que sean completos desde un punto de vista matemático.

   Haga un estudio comparativo de las consecuencias que para la teoría del conocimiento tienen las limitaciones expuestas por la máquina de Turing y por el teorema de Gödel.

Referencias para el capítulo 5

[BOUN72] Bourbaki, Nicolás, *Elementos de historia de las matemáticas*, Alianza Editorial, Col. Alianza Universidad, núm. 18, Madrid, 1972.
Un anónimo grupo de matemáticos franceses decidió recopilar con detalle la historia de las matemáticas y describirla desde una perspectiva formal. El autor Bourbaki en realidad no existe, es un seudónimo colectivo. Se han publicado ya varios volúmenes, y este libro es la traducción de un compendio de los primeros resultados.

[CHON57] Chomsky, Noam, *Syntactic Structures*, Mouton & Co., La Haya, Holanda, 1957.
Una de las primeras obras de este prolífico autor de lingüística, lingüística matemática y ciencias sociales. Este libro es "un intento de construir una teoría formalizada general de las estructuras lingüísticas, y de analizar su fundamentación". Propone

aquí su método de análisis basado en la estructura de las frases y las reglas de gramática transformacional.
Existe traducción al español.

[COHD86] Cohen, Daniel, *Introduction to Computer Theory*, Wiley, Nueva York, 1986.
Este es uno de los pocos libros para principiantes que se dedican por completo a la teoría de la computación. Está dividido en tres partes: teoría de autómatas, teoría de autómatas de pila (un tipo especial de autómatas, más generales), y "teoría de Turing". Bonito texto que resulta una completa y accesible fuente de teoremas y resultados de computabilidad.

[DEND78] Denning, Peter, Jack Dennis y Joseph Qualitz, *Machines, Languages and Computation*, Prentice Hall, New Jersey, 1978.
Libro de nivel avanzado sobre teoría de autómatas y computabilidad; además, contiene capítulos sobre funciones recursivas y sistemas formales de Post y sus equivalencias. Los dos primeros autores también son reconocidos como autoridades en sistemas operativos.

[DOUA70] Dou, Alberto, *Fundamentos de la matemática*, Nueva Colección Labor, núm. 117, Barcelona, 1970.
Traducción de un interesante y accesible libro sobre filosofía matemática. En él se describen algunas de las ideas que se han desarrollado acerca del porqué de las matemáticas, y se hace un recuento de las principales características de las escuelas de pensamiento matemático.

[GODK81] Gödel, Kurt, *Obras completas*, Alianza Editorial, Col. Alianza Universidad, núm. 286, Madrid, 1981.
Recopilación y traducción en un solo volumen de la obra completa de Gödel, que incluye sus teoremas fundamentales y muchos otros trabajos, todos ellos cortos, completamente densos y de muy alto nivel matemático. Se incluye un comentario de los traductores como introducción a cada trabajo.

[GRAF72] Gracia, Francisco (ed.), *Presentación del lenguaje*, Taurus, núm. 89, Madrid, 1972.
Compendio de traducciones de artículos sobre lingüística y aspectos de teoría de lenguajes. Contiene un largo artículo de Chomsky, así como uno titulado "Una demostración de la impracticabilidad de traducciones completamente automáticas y de alta calidad", escrito en 1960 por el investigador Yehoshua Bar-Hillel, pionero en la formalización de la lingüística.

[HEMN69] Hempel, C., E. Nagel, J. Newman, *et. al.*, *Matemática, verdad y realidad*, Grijalbo, Barcelona, 1974.

Traducción de un libro que recoge varios artículos descriptivos e introductorios sobre la función y la filosofía de las matemáticas que hacen varios autores, entre ellos los conocidos filósofos de la ciencia Carl Hempel y Ernest Nagel. Se incluye una explicación completa sobre el teorema de Gödel.

[HOPJ84] Hopcroft, John, "Turing Machines", en *Scientific American*, septiembre, 1984.
Interesante artículo sobre la máquina de Turing y los problemas de la computabilidad y los sistemas formales, tratado en forma introductoria por este autor, autoridad en el campo, y coautor también de numerosos textos sobre teoría de la computación y algoritmos, varios de los cuales se citan en este libro. Existe traducción al español.

[KLIM72] Kline, Morris, *Mathematical Thought from Ancient to Modern Times*, Oxford University Press, Nueva York, 1972.
Enciclopedia en la que se hace un recuento del saber matemático de la humanidad; tarea enorme que este reconocido autor de matemáticas lleva a cabo de una forma agradable y muy legible.

[MINM67] Minsky, Marvin, *Computation: Finite and Infinite Machines*, Prentice-Hall, New Jersey, 1967.
Este libro se considera ya un clásico en la literatura de alto nivel sobre computación y teoría de la computabilidad. Maneja desde estudios de corte filosófico hasta especificaciones de la teoría de las funciones recursivas, pasando por los autómatas y las máquinas de Turing. Minsky es también una de las autoridades sobre inteligencia artificial.

[PLAT73] Platón, *Diálogos*, Porrúa, México, 1973
Cualquiera que lea algunos de los clásicos diálogos de Platón comprenderá por qué se incluyen en la bibliografía del capítulo donde se tratan temas de computabilidad y lógica, ya que resulta evidente que buena parte de los fundamentos de nuestros métodos de conocimiento se encuentran expuestos en esta obra de hace más de 20 siglos.

6

Elementos de programación

6.1 Introducción

Una vez entendidos los principios elementales de funcionamiento de un sistema de cómputo y estudiado el concepto fundamental de algoritmo, pasaremos a una sección más operativa dentro de este curso: aprender a programar una computadora. Se intenta que el lector adquiera algunas habilidades prácticas que le permitirán, con la atención suficiente, escribir programas para computadora correctos, legibles y claros.

Cabe hacer una advertencia: se aprenderá a *programar* una computadora, no a *codificar* un programa. La diferencia entre ambos conceptos es fundamental, y no está entendida del todo en el medio profesional de la computación ni por completo, por desgracia, en el medio académico.

Diferencias entre programación y codificación

Por *programar* se entiende un proceso mental complejo, dividido en varias etapas. La finalidad de la programación, así entendida, es comprender con claridad el problema que va a resolverse o simularse por medio de la computadora, y entender también con detalle cuál será el procedimiento mediante el cual la máquina llegará a la solución deseada.

La *codificación* constituye una etapa necesariamente posterior a la programación, y consiste en describir, en el lenguaje de programación adecuado, la solución ya encontrada, o sugerida, por medio de la programación. Es decir, primero se programa la solución de un problema y después hay que traducirla a la computadora.

La actividad de programar es, más que nada, conceptual, y su finalidad es intentar definir, cada vez con mayor precisión, acercamientos que resuelvan el problema de manera virtual, es decir, que efectúen una especie de "experimentos mentales" sobre el problema por resolver o simular. El resul-

tado de tales experimentos constituirá una descripción de los pasos necesarios para encontrar la solución.

Esta descripción, como cualquier otra, estará expresada en un lenguaje determinado. La importancia de la programación consiste en que este lenguaje funciona a la vez como vehículo descriptor y como modelo de la representación dada a la solución; las principales características de ese lenguaje son que es neutro y completo. El primer concepto denota su independencia respecto a alguna máquina en particular, y el segundo se refiere al poder del mismo para expresar cualquier idea computacional.

Un lenguaje así recibe el nombre de **pseudocódigo**, y nuestra primera tarea consiste en entenderlo y aprenderlo, para después aplicarlo a tareas sencillas de programación.

6.2 Fases de creación de un programa

Escribir programas para computadora es una actividad que requiere una buena cantidad de esfuerzo mental y de tiempo. Armados ya de una descripción teórico-conceptual de lo que es una computadora y lo que es un algoritmo, emprendemos el camino de volver realidad la construcción de modelos para representar descripciones de fenómenos o procesos del mundo real.

Esto implica una metodología científica, repetible y comprobable. Se hablará ahora del proceso mental asociado a la construcción de programas para una computadora. Se observará que las primeras fases de este proceso no difieren demasiado de las equivalentes para casi cualquier otra rama de las ciencias básicas, en el sentido de que constituyen un conjunto de pasos bien especificados que se acercan paulatinamente a una solución.

Las fases en la construcción de un programa son, en orden, las siguientes (aunque debe quedar claro que no hay límites tajantes entre el final de una y el inicio de otra*):

0) *Entender* el problema.
1) Hacer el *análisis* del mismo (a veces este paso se denomina análisis del sistema).
2) *Programar* el modelo de solución propuesto.
3) *Codificarlo*.
4) Cargarlo a la computadora para su *ejecución y ajuste*.
5) Darle *mantenimiento* durante su tiempo de vida.

El paso cero parece banal, pero deja de parecerlo cuando se piensa en la gran cantidad de proyectos de computación que se desarrollan (y a veces se terminan) sin haber comprendido bien para qué eran, o cuál era el problema

* Compárense estas fases con las propuestas en [TREJ82] (págs. 351-353), por ejemplo.

que supuestamente iban a resolver. Y si, además, se toma en cuenta que los sistemas de programación reales, a diferencia de los ejercicios de carácter didáctico o académico, muchas veces son largos y complejos, e implican la participación de varias personas (a veces decenas o cientos) durante largos periodos, se podrá comprender la importancia de entender con claridad el problema antes de abocar recursos a su solución. El mundo está demasiado poblado de proyectos y sistemas (no sólo de computación) que o no resuelven el problema para el que fueron diseñados, o bien lo hacen pero de una manera sólo aproximada y deficiente*.

No existe un criterio único e infalible para entender con claridad un problema, por lo que nos conformaremos con recomendar mesura y claridad en el momento de enfrentarse con uno por vez primera. Al final, de lo que se trata es de crear y mantener una idea clara, un "mapa mental" del problema propuesto, y de ser capaz de abarcarlo de un solo vistazo. Esto obliga a hacer caso omiso de detalles y particularidades operativas en una primera instancia.

Análisis de sistemas

La segunda fase —primera para nosotros— es muy importante; consiste en efectuar un análisis completo del problema o sistema existente, con la finalidad de proponer un modelo para su solución. Está claro que este modelo no puede existir sin que se hayan especificado con claridad todos y cada uno de los componentes estructurales del sistema.

Entendemos por sistema un conjunto estructurado de elementos interrelacionados de alguna manera que puede hacerse explícita. Obsérvese que para la definición del sistema no nos preocupa la función que éste desempeña, ni tampoco exigimos a sus componentes que cooperen entre sí para conseguirlo, ni ninguna otra consideración de carácter animista o teleológico.

Concepto de sistema

Se insiste en los aspectos estructurales porque son la clave para entender y analizar un problema no trivial.

La *estructura* de un sistema es la forma en que están relacionados entre sí sus diversos componentes, de modo que es perfectamente posible tener dos sistemas diferentes con componentes iguales. La diferencia estará en la forma de hacer corresponder unos con otros.

Éste no es el lugar para discutir acerca de lo que son los sistemas, pero sí cabe decir lo que no son. Por desgracia, existe una concepción, demasiado ligera y vulgarizada, de que cualquier cosa puede suponerse como un siste-

* Claro que en este lamentable estado de cosas influyen muchos otros aspectos, además de no entender con claridad el problema antes de intentar su solución, y casi todos ellos tienen que ver con consideraciones de corte psicológico —y sociológico—. Un estudio muy original de algunas de las causas de que las cosas salgan mal se puede encontrar en el divertido libro *El principio de Peter*, de Laurence Peter y Raymond Hull, Plaza Janés, Barcelona, 1971. El inglés Northcote Parkinson se dedicó durante los años de la Segunda Guerra Mundial a estudiar las causas del mal funcionamiento de las organizaciones, y esto dio como resultado las conocidas "leyes de Parkinson", acerca de las cuales escribió varios libros posteriormente. En los ambientes de ingeniería son muy conocidas también las "leyes de Murphy", que intentan explicar en una o dos frases la razón de que los proyectos no funcionen como se esperaba.

ma, y ha surgido en consecuencia una fiebre de considerar todo desde "el punto de vista de los sistemas". Los peligros de tal enfoque son graves, pues pueden llevar a generalizaciones carentes de rigor metodológico y científico. Hay que tener cuidado de llamar sistema solamente a aquellos complejos de elementos en los que su interrelación pueda ser explicitada por medio de un modelo matemático o, por lo menos, a partir de una descripción libre de ambigüedades.

Volviendo a nuestro punto, se debe efectuar el análisis del sistema (o problema) que es candidato a ser "computarizado". Para esto se dispone de varios enfoques cualitativos, cuya finalidad consiste en proponer el lugar y la función de los componentes aislables del sistema, en términos tanto de los demás como de la función, ahora sí, que será desempeñada por el conjunto.

Un primer enfoque consiste en aplicar una especie de "rejilla mental" y superponerla al sistema, de manera que sus componentes queden englobados en algún elemento de ella. Esta no es otra cosa que una manera estándar de atacar problemas y darles soluciones preestablecidas por la práctica. Este procedimiento se emplea muy a menudo en problemas rutinarios, y propone soluciones que funcionan adecuadamente para la mayoría de los casos tipificables. En ingeniería civil es común aplicar métodos preestablecidos en manuales para resolver problemas de estructuras sencillas. Siempre que un método de esta clase cumpla sus objetivos no deberá haber impedimento para usarlo, pero son realmente pocos los casos complejos donde puede resultar de utilidad.

Existe al menos otro método para problemas dotados de una estructura menos normalizable, y consiste en formular un modelo que se adapte especialmente a la "forma" del problema. Así debe ser, porque se parte de que el problema en estudio no es tipificable; esto significa, en forma figurada, que no tiene una forma preestablecida pues, si la tuviera, habría una entrada en el manual correspondiente que la describiera, junto con una solución estandarizada.

La función del analista de sistemas consiste precisamente en describir el modelo que mejor se adapte a la estructura del problema que se estudia. Un enfoque funcional puede ser adecuado en muchos casos (es decir, hacer el análisis partiendo de la función que cada componente desempeña en el sistema como un todo). Sin embargo, en otro tipo de problemas puede emplearse un análisis dirigido por los datos que maneja el sistema, o por algún otro aspecto que pueda servir de guía.

Función del análisis

El análisis de sistemas en computación es una actividad compleja y altamente dependiente de consideraciones humanas; por tanto, no ha sido aún comprendida en su totalidad dentro de un esquema matemático. Esto quiere decir que las experiencias previas en el análisis son factor primordial en el desarrollo de uno nuevo, y que no existe —a nuestro entender— una manera "segura" de lograr un análisis correcto o productivo en primera instancia, sino que el proceso está sujeto a mejoras, que pueden ser producto de esquemas inductivos o de simples ensayos de prueba y error.

El resultado final de un análisis puede consistir en diagramas que muestren el flujo de la información (no confundirlos con los diagramas de flujo).

Esto es equivalente a un mapa que muestra los diferentes caminos que la información toma dentro del conjunto, junto con una jerarquización de las diversas funciones que el sistema desempeña. El resultado del análisis también puede ser una descripción del funcionamiento del sistema actual (en caso que exista), o de cómo se propone que funcione el nuevo propuesto.

Más adelante se dan ejemplos de pequeños análisis, que desembocarán en programas completamente terminados.

Programación

Una vez hecho el análisis de un sistema se procede a convertirlo en un programa de computadora, que estará escrito en pseudocódigo.

Tal vez el siguiente ejemplo nos acerque a la idea que se desea: pensemos en los procesos que pasan por la mente de un arquitecto cuando conoce el terreno, posiblemente lleno de desniveles y rocas, donde habrá de construir un edificio. ¿no tiene acaso que imaginarse la obra completa y hacer un considerable esfuerzo de abstracción, sin el cual será poco provechoso el trabajo posterior? Éste es el papel que cumple el análisis ya descrito, y que ahora deberá verse plasmado y cristalizado en la forma de un programa.

Un programa está formado, estructuralmente, por dos tipos de componentes: estructuras de control y estructuras de datos. Las siguientes secciones estudian sus características, su ciclo de vida, y su funcionamiento y desarrollo.

Las **estructuras de control** son las formas que existen para dirigir el flujo de acciones que el procesador efectuará sobre los datos que se manejan en un programa, mismos que están organizados de maneras diversas que son, precisamente, las **estructuras de datos**.

Las estructuras de control básicas, que se estudian más adelante, son la secuenciación, la selección y la iteración condicional, mientras que las estructuras de datos más comunes son los arreglos (o vectores), listas, cadenas, pilas y árboles, que también se emplean —aunque no se estudian con profundidad— a lo largo de los siguientes capítulos.

Como componentes no estructurales de un programa se puede mencionar, en orden creciente de complejidad, los enunciados, instrucciones, funciones, subrutinas (o procedimientos) y módulos. De todos ellos se hablará en el transcurso de este capítulo y en el siguiente.

Llamar no estructurales a los anteriores elementos significa que su aparición dentro de un programa obedece a razones guiadas por los componentes que sí lo son: las estructuras de datos y de control. O lo que es igual, lo primero que hay que definir al construir un programa son precisamente sus elementos estructurales, aquellos sin los cuales el programa no es tal.

Y éste es el momento de repetir lo que se dijo en la introducción del curso: se aprenderá primero a programar, no a codificar; nos preocupará un problema estructural y no simplemente instancias particulares por resolver, por lo que no será sorpresa encontrarse ahora con una descripción teórico-

conceptual acerca de la programación. No comenzaremos a preocuparnos de los detalles, siempre enfadosos, de los lenguajes de programación específicos.

De lo que se trata ahora es de definir lo que es un **programa**, y se propone lo siguiente: un programa es un conjunto de **declaraciones** de estructuras de datos, seguidas de un conjunto de **proposiciones** (usando esta palabra en un sentido amplio, que abarca todos los componentes de las estructuras de control). Además, este programa o cadena de símbolos válidos cumple otra condición: está bien formado. Una cadena bien formada (o bien construida) es aquella que está hecha siguiendo las reglas sintácticas (en el sentido definido cuando se habló de los compiladores) de la gramática que produce el lenguaje de computación que se emplea.

Como en este caso no se está hablando de ningún lenguaje de programación en particular, entonces nos referiremos a un programa bien formado cuando sea el producto de la aplicación de ciertas reglas de construcción primitivas. Más adelante se muestra que por medio de estas reglas es posible construir cualquier programa que, por tanto, estará bien formado.

Codificación

Una vez terminada la fase de programación se habrá producido una descripción del modelo propuesto, escrita en pseudocódigo. La razón de ser de ese paso fue disponer de un programa que pueda ser probado mentalmente para averiguar si es correcto en principio, y para determinar en qué grado considera todo el análisis hecho anteriormente. El proceso mediante el cual se llega a un programa esencialmente correcto recibe el nombre de refinamientos progresivos y será estudiado con detalle más adelante.

Sin embargo, un programa en pseudocódigo no es ejecutable en una computadora, por lo que se requiere refinarlo más. El objetivo de estos refinamientos consiste en acercar lo más posible el programa escrito en pseudocódigo a un programa escrito en algún lenguaje de programación particular (como los descritos en el anexo de este capítulo). Esta fase, necesariamente posterior a la de programación, se trata ampliamente en el capítulo 8, con los lenguajes FORTRAN y Pascal. El artículo [WIRN84] muestra en forma concisa la relación que debe existir entre un algoritmo y las estructuras de datos que requiere.

Ejecución y ajuste

Cuando al fin se tiene el programa codificado y compilado llega el momento de ejecutarlo y probarlo "sobre la marcha". Es decir, permitir que la computadora lo ejecute para evaluar los resultados.

La nociva práctica usual —que tiende a desaparecer— es dedicar poco tiempo a las etapas de análisis y programación y enfocar la atención y los recursos a la codificación, razón por la cual la ejecución de uno de tales progra-

mas estará casi siempre plagada de errores. Existen dos tipos de fallas que es posible encontrar en un programa ya codificado: errores de sintaxis y errores de lógica de programación. Los primeros son relativamente triviales, mientras que los segundos son los causantes de los frecuentes retrasos que sufren los proyectos de programación en todos los niveles de complejidad.

Tipos de errores

En efecto, un error de lógica apunta claramente a omisiones y errores en el modelado que se está tratando de hacer de la realidad. Esto casi siempre se debe a un deficiente análisis o a una programación en pseudocódigo incompleta y apresurada. El grueso del trabajo creativo debe dedicarse precisamente a las etapas que planean y hacen posible la codificación.

La concepción moderna de la prueba de un programa se ha desplazado de la etapa de ejecución a la etapa de programación en pseudocódigo, con las consiguientes ventajas en ahorro de recursos de cómputo utilizados y de tiempo dedicado al cansado ciclo (que a veces parece sin fin) de codificación-compilación-ejecución-corrección-codificación.

Esto no implica, por supuesto, que la metodología propuesta sea infalible o produzca resultados limpios en la primera prueba; significa, sí, que el camino que lleva de la concepción de un sistema hasta su ejecución por medio de una computadora sea más corto y con menos sobresaltos.

Mantenimiento

Si se ha tomado el trabajo de planear cuidadosamente un sistema y de transformarlo en un conjunto bien estructurado de programas y módulos, seguramente tendrá una vida útil prolongada y no se utilizará sólo una o dos veces. Este simple hecho obliga a considerar un esquema de mantenimiento que asegure que el modelo ya sistematizado evolucione a un ritmo parecido al que lo haga la realidad que está siendo simulada. Tal vez llegue el momento en que ese proceso o aspecto de la realidad para el que se construyó el sistema haya cambiado cualitativamente, en cuyo caso se habla del término de la vida útil del sistema.

Vida útil de un sistema

Mientras tanto, sin embargo, hay que ser capaces de hacer alteraciones no estructurales al sistema con costo mínimo en recursos de análisis y programación, lo cual de alguna manera está asegurado si el sistema se ha construido de manera modular y estructurada, y si se dispone de la documentación adecuada que lo describa tanto en su diseño como en su uso. Suele decirse que si un sistema sólo es comprensible por su creador, es un mal sistema.

Esa falta de flexibilidad, además, resulta imposible de tolerar para el caso de sistemas realmente grandes, que son creados incluso por cientos de ingenieros en sistemas y programadores. Se dice, por ejemplo, que el sistema operativo de la serie 360 de IBM requirió cerca de 5000 años-hombre para su desarrollo. No puede ser que un sistema de tal magnitud no tenga previstos cambios y adaptaciones constantes.

Aunque nuestros programas no sean siquiera medianamente grandes, tenemos el compromiso de hacerlos claros y flexibles para que admitan mejoras o sugerencias posteriores.

Ha llegado ya el tiempo de poner manos a la obra y aprender los fundamentos de la programación.

6.3 Herramientas para construir programas

En esta nueva sección se mostrarán algunas de las herramientas existentes para escribir programas, una vez que el problema se ha comprendido y que se ha efectuado un análisis, aunque sea elemental, del mismo.

La primera tarea consiste simplemente en escribir en español una descripción de los pasos necesarios para resolver el problema, con base en el análisis previamente efectuado. A continuación vendrá la fase de traducir esto usando ciertas herramientas computacionales que ahora se describen.

Lo primero que resalta en una descripción de esta clase es su carácter secuencial. Se inicia con el primer paso, se continúa con el segundo, y así hasta llegar al fin. Por trivial que pueda parecer esta idea siempre debe tenerse como estandarte en la programación. Es decir, que la programación es necesariamente una actividad ordenada y disciplinada, que exige en todo momento una gran cohesión en las actividades mentales tendientes a describir adecuadamente el problema que se desea modelar.

Secuenciación

Pero no todos los procesos pueden ser descritos sólo con la primera herramienta mencionada, la **secuenciación**. Se hace necesaria una que permita tomar decisiones sencillas. Esta nueva construcción primitiva se llama **selección**, y consiste en evaluar, durante la ejecución, una condición booleana, para decidir cuál de dos caminos escoger a continuación. Una condición se llama *booleana** cuando puede adquirir únicamente dos valores de verdad: falso o verdadero.

Cualquier pregunta que admita tan sólo dos posibles respuestas (sí o no) es booleana. Por ejemplo, "¿Cuántos años tienes?" no es una pregunta booleana, puesto que admite cerca de cien posibles respuestas. Pero la pregunta "¿Tienes 25 años?" sí lo es, puesto que solamente puede ser respondida con un sí o un no.

Otra característica necesaria para que una pregunta (o condición) sea booleana es que sus dos posibles respuestas sean mutuamente excluyentes. Esto es, por el principio del tercero excluido, o tengo 25 años o no los tengo, pero no existe una posibilidad intermedia.

Selección

La construcción primitiva de selección, entonces, dirá algo así: "Evalúa cierta condición booleana. Si el resultado es verdadero, entonces ejecuta la proposición 1; en otro caso, ejecuta la proposición 2".

Una tercera y última construcción primitiva completará un conjunto que ha demostrado ser completo, en el sentido de que con él es posible escribir

* En honor a George Boole, matemático inglés que propuso todo un sistema de lógica formal que se analizó en el anexo 5.4.

cualquier programa. Esta nueva estructura de control recibe el nombre de **iteración condicional**. Consiste, como su nombre lo indica, en el planteamiento de una repetición de acciones, gobernada también por una condición booleana.

Iteració.. condicional

Una iteración condicional dice: "Evalúa cierta condición booleana. Si el resultado es verdadero, ejecuta una proposición y continúa de esta manera mientras la condición siga siendo verdadera". Está claro que el ciclo se romperá cuando la condición deje de ser verdadera y se vuelva falsa. Es posible, incluso, que el ciclo nunca se efectúe, si desde el principio la condición es falsa.

La proposición más simple sobre la que pueden trabajar las tres estructuras fundamentales de control es el **enunciado**. Un enunciado será la unidad mínima que que se pueda ejecutar. Por ejemplo, se quiere lograr que la variable ALFA adquiera el valor cinco (véase pág. 80), esto se puede representar con el siguiente enunciado (que es en realidad una instrucción de **asignación**): `ALFA = 5`.

Otro enunciado puede ser, por ejemplo, "escribe el valor actual de la variable ZETA": escribe ZETA, que en realidad es una instrucción de entrada/salida.

Representaremos un enunciado cualquiera por medio de la letra *e*, y cuando haya necesidad de diferenciar entre varios, entonces se numerarán: $e1$, $e2, \ldots, en$.

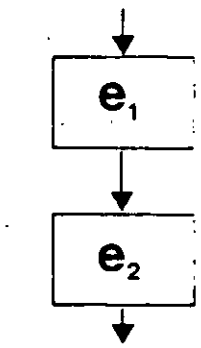
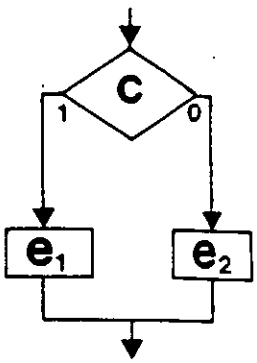
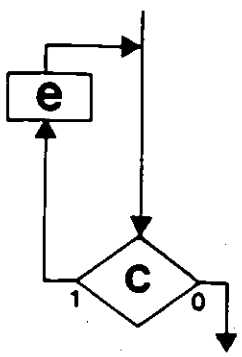
Por otro lado, una condición booleana se representará con la letra mayúscula *C*. Cuando haya que diferenciar entre varias, también serán numeradas. A veces en lugar de un sí se empleará *V* (de verdadero) o el dígito 1 . En lugar del no podrá usarse *F* (de falso) o el dígito 0.

Existen por lo menos dos maneras de representar las estructuras de control ya descritas, junto con los enunciados y las condiciones booleanas. Una de ellas es gráfica, mediante esquemas llamados *diagramas de flujo*. La otra es por medios simbólicos, usando pseudocódigo.

Esta última forma, que será la que emplearemos a lo largo de este capítulo y el siguiente, requiere de ciertos símbolos privilegiados, que ya tienen significado preciso y establecido de antemano. A tales indicadores del pseudocódigo se les conoce como **palabras clave**. Es necesario que exista una palabra clave para la selección y otra para la iteración condicional, y para instrucciones adicionales y otras estructuras de control que veremos después. Por ejemplo, la palabra escribe que se usó líneas atrás es una palabra clave que ya tiene significado preestablecido, a diferencia de la palabra ALFA, que es una variable libre.

En virtud de que las palabras clave son palabras que hablan acerca de otras, adquieren la categoría de *metapalabras*, razón por la cual se deben distinguir de las que no lo son, para lo que se subrayan.

A continuación se describen las tres estructuras de control, empleando las dos formas ya explicadas:

Estructura de control	Pseudocódigo	Diagrama de flujo
SECUENCIACIÓN	*e1* *e2* o bien *e1*; *e2*	
SELECCIÓN	si *C* entonces *e1* otro *e2*	
ITERACIÓN CONDICIONAL	mientras (*C*) *e*	

Se pueden observar varias cosas importantes. Existen palabras reservadas para la selección y la iteración condicional, mas no para la secuenciación, porque en tal caso basta simplemente con escribir los enunciados: uno en cada renglón, o en el mismo, pero separados por un punto y coma.

En el pseudocódigo de la selección decidimos aprovechar los espacios en blanco del papel para dibujar la estructura, poniendo en renglones independientes, pero en la misma columna, las dos partes que son mutuamente excluyentes. Esta es una ventaja de la representación en pseudocódigo, ya que permite determinar de inmediato la estructuración del programa fuente.

Una sola entrada-una sola salida

Una observación primordial es que cada una de estas estructuras de control tiene un solo punto de entrada y un solo punto de salida. Esto servirá para armar programas completos uniendo salidas con entradas, para formar cadenas de proposiciones estructuradas en secuencia.

Por ejemplo, en el pseudocódigo de la secuenciación, la entrada es la proposición anterior al renglón que comienza con el enunciado e_1 (que no se ve), y la salida es la proposición que está después (que tampoco aparece). Igual sucede con las otras dos.

En los diagramas de flujo la entrada estará arriba del dibujo y la salida estará abajo (aunque, como se verá más adelante, los conceptos de arriba y abajo en los diagramas de flujo son bastante engañosos).

Como ya se dispone de estructuras básicas de control, se procederá a hacer uso de ellas, combinándolas de diversas maneras para escribir programas completos. Para este fin existe una *regla fundamental de composición*, que dice lo siguiente:

> Es posible combinar las estructuras de control de secuenciación, selección e iteración condicional, utilizando para tal fin la secuenciación, la selección y la iteración condicional.

Es decir, es posible hacer, por ejemplo, la secuenciación de una secuenciación con una secuenciación, o la selección entre una secuenciación y una iteración condicional.

Si esta regla de composición parece rara es porque es recursiva. Obsérvese que define nuevas estructuras de control (de cualquiera de los tres tipos conocidos) a partir, precisamente, de los tres tipos de estructuras de control.

Como se decía seis palabras atrás, una definición es recursiva si emplea el *definiens* dentro del *definiendum* o, en otras palabras, si dice cómo obtener conceptos nuevos ¡empleando para tal fin el mismo concepto que se intenta definir! Este razonamiento parece fallido; parece que no llevará a ninguna parte por ser circular. En realidad, los razonamientos recursivos se encuentran en la base misma de las matemáticas, porque son necesarios para describir conceptos centrales, como el de número.

Un razonamiento recursivo tiene dos partes: la base y la regla recursiva de construcción. La base no es recursiva y es el punto tanto de partida como de terminación de la definición.

El conjunto de los números naturales puede definirse recursivamente así:

> Base: El 1 es un número natural. Todo número natural tiene un sucesor.
> Regla: El sucesor de un número natural es un número natural*.

Es decir, la base es la existencia del número natural 1. Como ese número existe, debe tener un sucesor (que se llama 2). Por tanto, 2 existe, y como ya es un número natural, entonces debe tener un sucesor, y así, *ad infinitum*.

---

* Esto fue expresado, aunque no exactamente de esta forma, por el gran matemático y lógico italiano Giuseppe Peano (1858-1932).

Regla de formación de programas

Podría reconsiderarse la regla de formación de estructuras válidas de control de la siguiente manera:

Base: La secuenciación, la selección y la iteración condicional son estructuras válidas de control que pueden ser consideradas como enunciados.

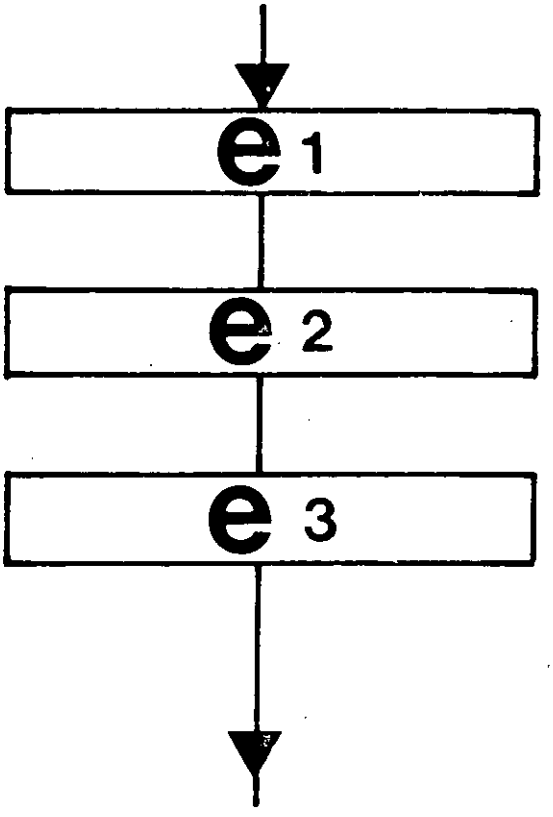
Regla: Las estructuras de control que se puedan formar combinando de manera válida la secuenciación, selección e iteración condicional también serán válidas.

Esto da origen al concepto de **programación estructurada**. Un programa estará bien construido si está formado por estructuras de control válidas, de acuerdo con la regla precedente. Las referencias [DAHO72], [MCGC75] y [LINR79] tratan, al igual que parte de nuestro próximo capítulo, de la teoría de la programación estructurada*.

Ya sabemos que la secuenciación de dos enunciados (e_1 y e_2) se forma así: e_1; e_2. ¿Cómo se formará la de tres? Pues con una secuenciación de la secuenciación anterior (considerada como un único enunciado compuesto) con el nuevo enunciado e_3, para entonces obtener: e_1; e_2; e_3.

Obsérvese que esta nueva formación sigue teniendo una sola entrada y una sola salida, puesto que es también una secuenciación.

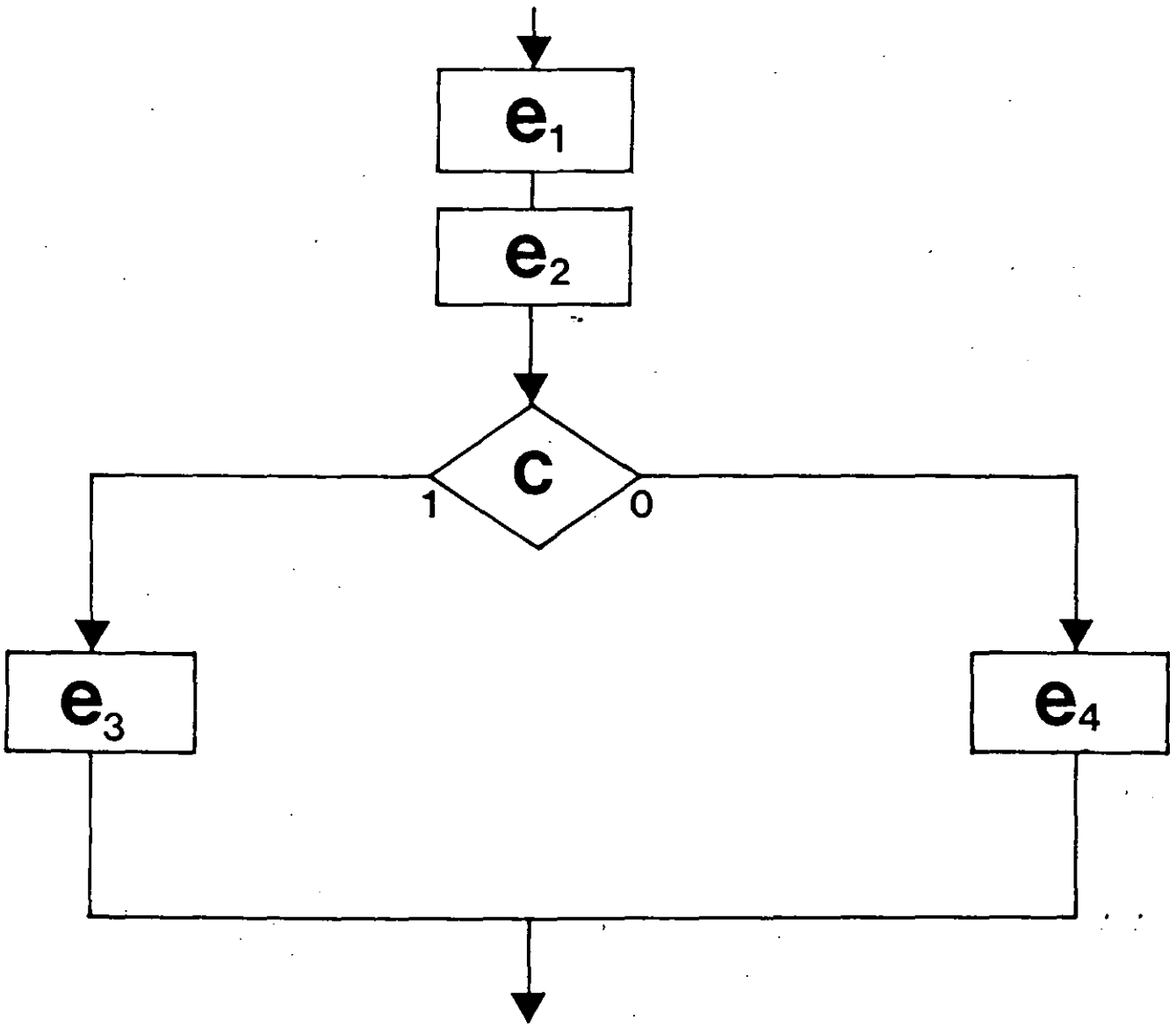
* Los antecedentes teóricos de la programación estructurada se remontan a los inicios de la teoría de la computabilidad, y uno de los primeros resultados, que proponía métodos de normalización para diagramas de flujo, se expuso en un artículo comúnmente considerado el pionero en este campo, "Flow Diagrams, Turing Machines and Languages With Only Two Formation Rules", de Conrado Bohm y Giuseppe Jacopini, publicado en *Communications of the Association for Computing Machinery*, vol. 9, núm. 5, mayo de 1966. El artículo, sin embargo, es completamente teórico, y no tiene una conexión directa con las tareas de programación, pero aun así es señalado como el más importante, y como el que dio origen a lo que se conoce como el "Teorema de la programación estructurada". En el artículo "On Folk Theorems", de David Harel, aparecido también en *Communications of the ACM*, vol. 23, núm. 7, julio de 1980, se hace una incisiva y mordaz crítica a toda esta concepción, un tanto mítica, de los orígenes de la programación estructurada. Un poco para contrarrestar esta visión mitificada de la programación estructurada (y de sus "axiomas"), Donald Knuth escribió un extenso artículo titulado "Structured programming with go to statements", publicado en *Computing Surveys of the ACM*, vol. 6, núm. 4, 1974, en el que explica, con ejemplos, cómo sí es posible (y hasta deseable en algunos casos) emplear la proposición *go to* con provecho dentro de la programación estructurada.



Obtengamos ahora la secuenciación de una secuenciación con una selección:

```
e1
e2
si  C  entonces  e3
       otro e4
```

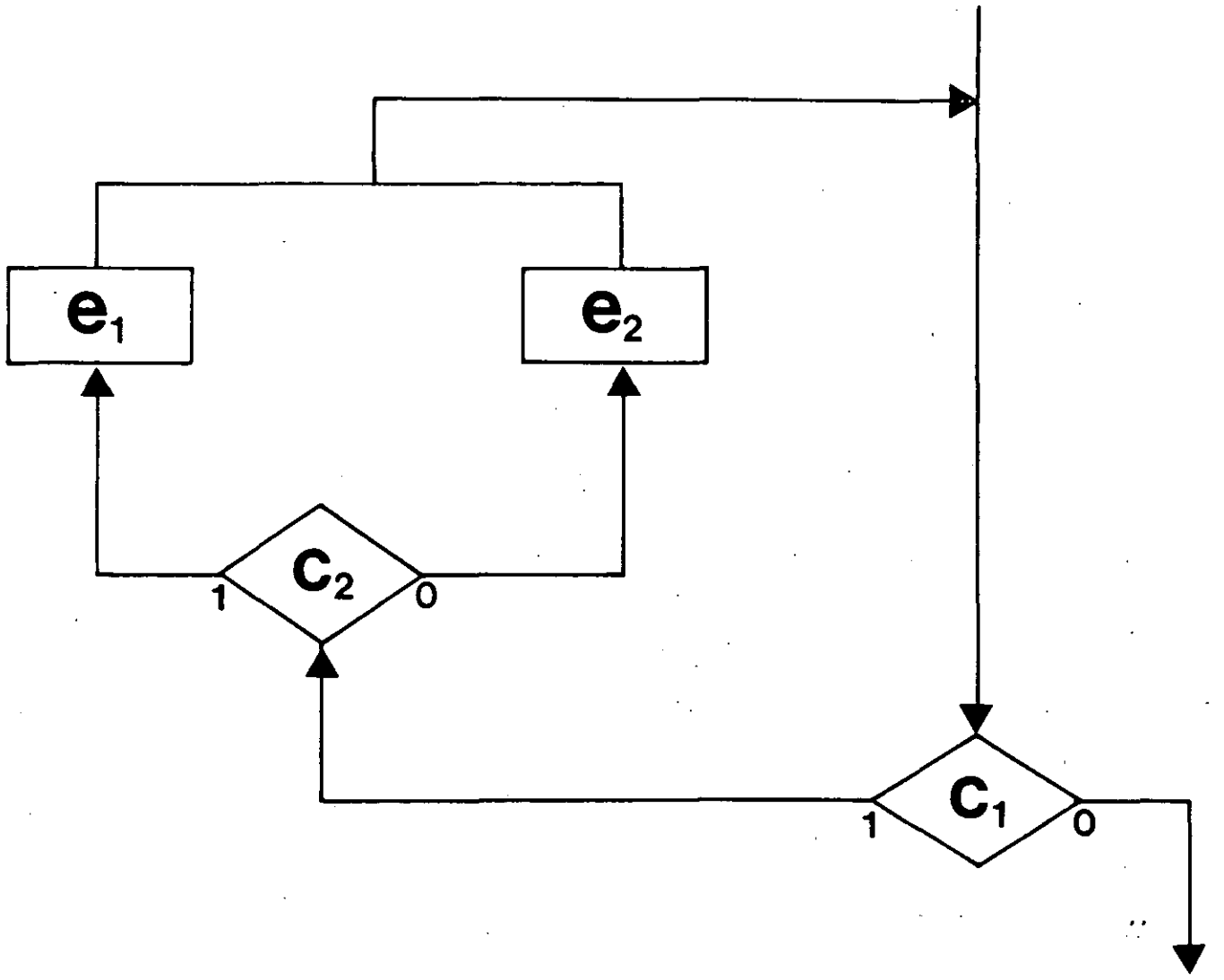
La nueva estructura (secuenciación compuesta) sigue teniendo una sola entrada y una sola salida, y éste es su diagrama de flujo:



Debe estar claro ya que esta otra también es válida, porque se trata de la iteración condicional de una selección:

```
mientras ( C1 )
si  C2  entonces e1
        otro e2
```

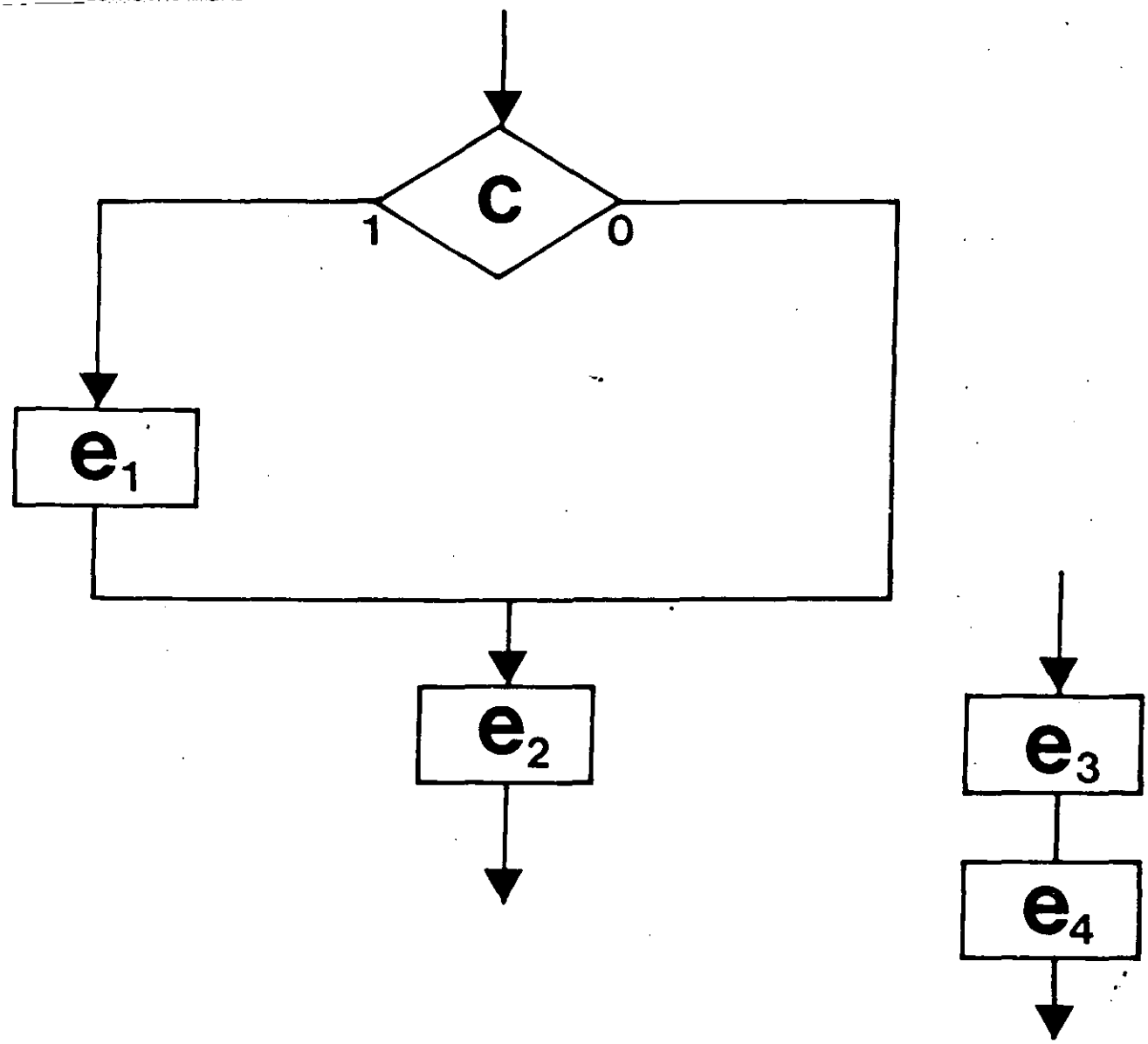
que tiene el siguiente diagrama de flujo:



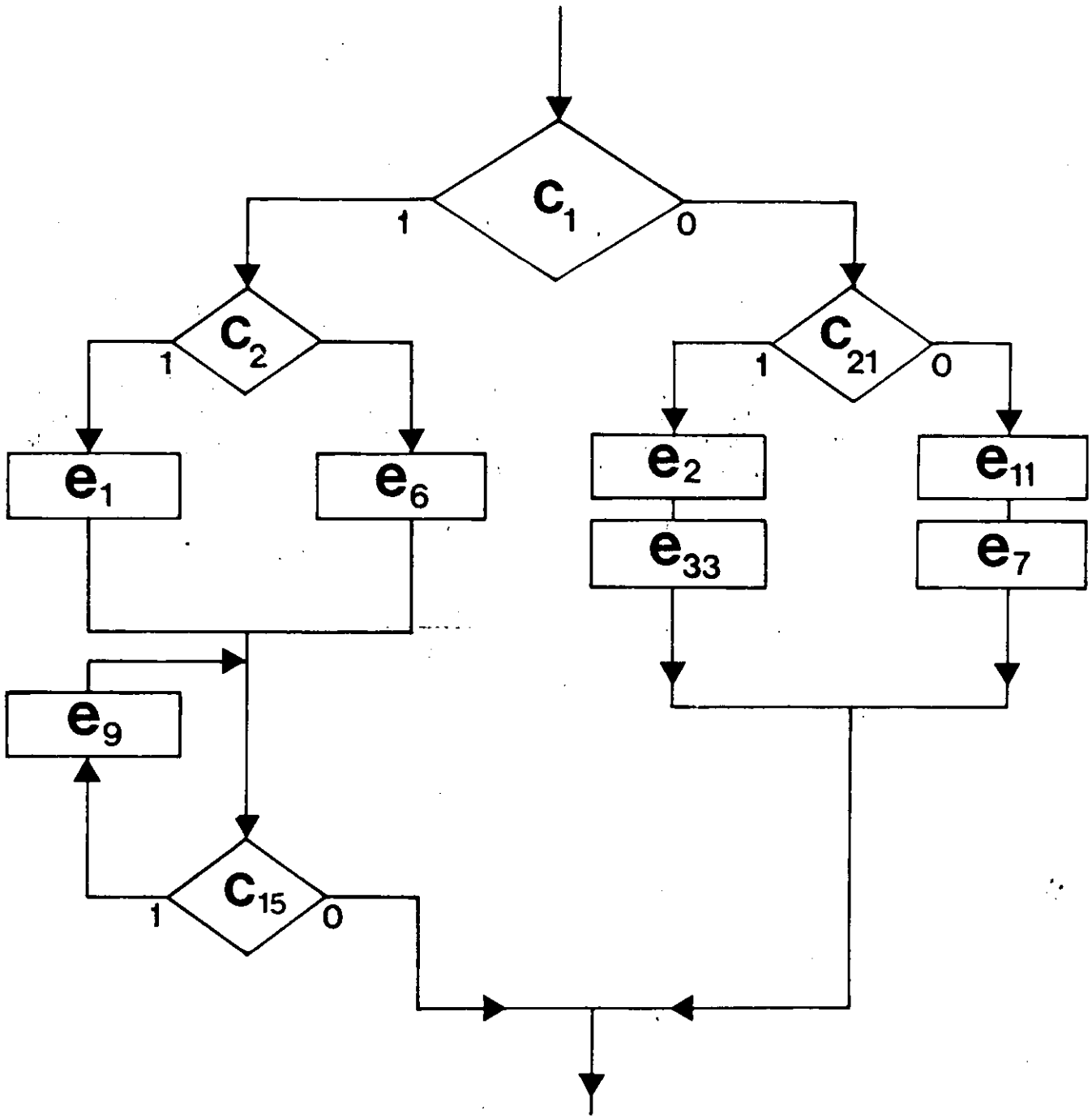
Ahora intentemos hacer la selección de una secuenciación con otra:

```
si  C  entonces  e1; e2
       otro e3; e4
```

Aquí tenemos un problema de ambigüedad: no está claramente definido el alcance del entonces, ni el alcance del otro. O sea, no está claro si *e2* se ejecuta después de *e1* solo si *C* es verdadera o si, por el contrario, *e2* se ejecuta después de *e1* sin preocupar el valor de *C*. Si se escoge la segunda posibilidad, el otro está fuera de lugar, porque la estructura quedaría definida así:



en lugar de quedar de esta otra manera, que es la correcta:



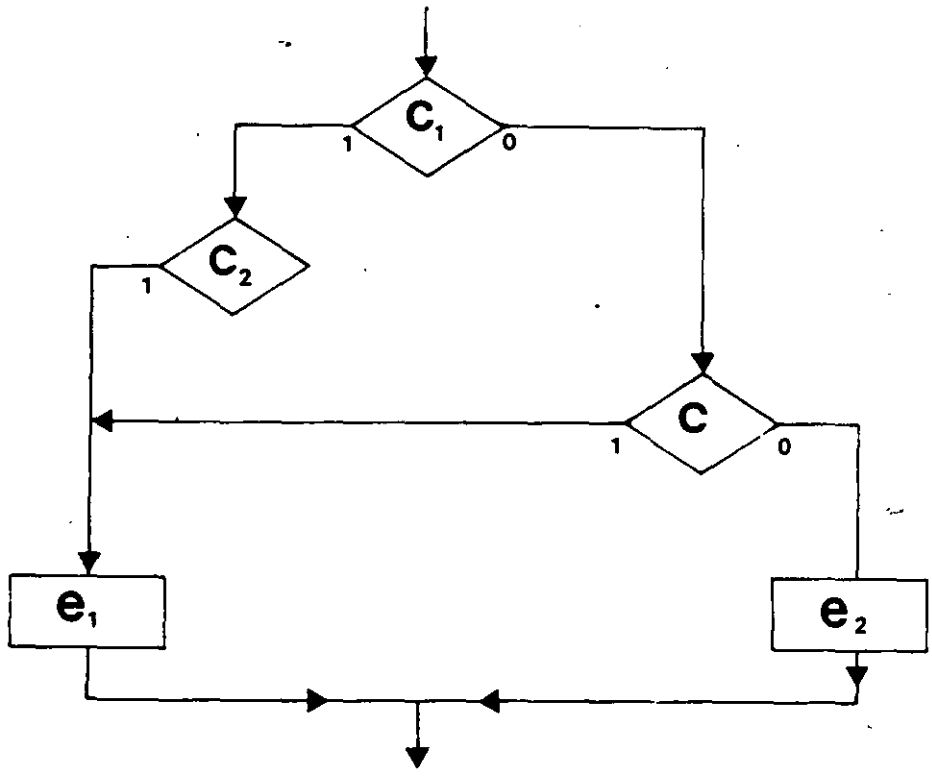
El problema con los diagramas de flujo —que no recomendamos— es que a medida que crece la complejidad (grado de anidamiento) de las proposiciones, también crece el detalle con el que hay que dibujarlos. Esto llega a convertirlos en figuras fraccionadas (pues de otro modo no habría espacio suficiente en la hoja) difíciles de seguir y entender. Además, cuando el diagrama es complejo y tiene proposiciones de tipo **mientras**, que a veces obligan a que

la entrada y la salida del enunciado asociado estén dibujadas "de cabeza", entonces sí que resultan oscuros.

Pruebe el lector hacer ejercicios de construcción y anidamiento de estructuras de control para que se dé cuenta por sí mismo.

Si está bien formada, tiene una sola entrada y una sola salida

Si una estructura de control compleja está bien formada (estructurada), entonces seguirá teniendo una sola entrada y una sola salida. Pero la conversa no es cierta. O sea, el que una figura tenga una sola entrada y una sola salida no necesariamente significa que esté bien formada, como lo demuestra el siguiente diagrama de flujo no estructurado:



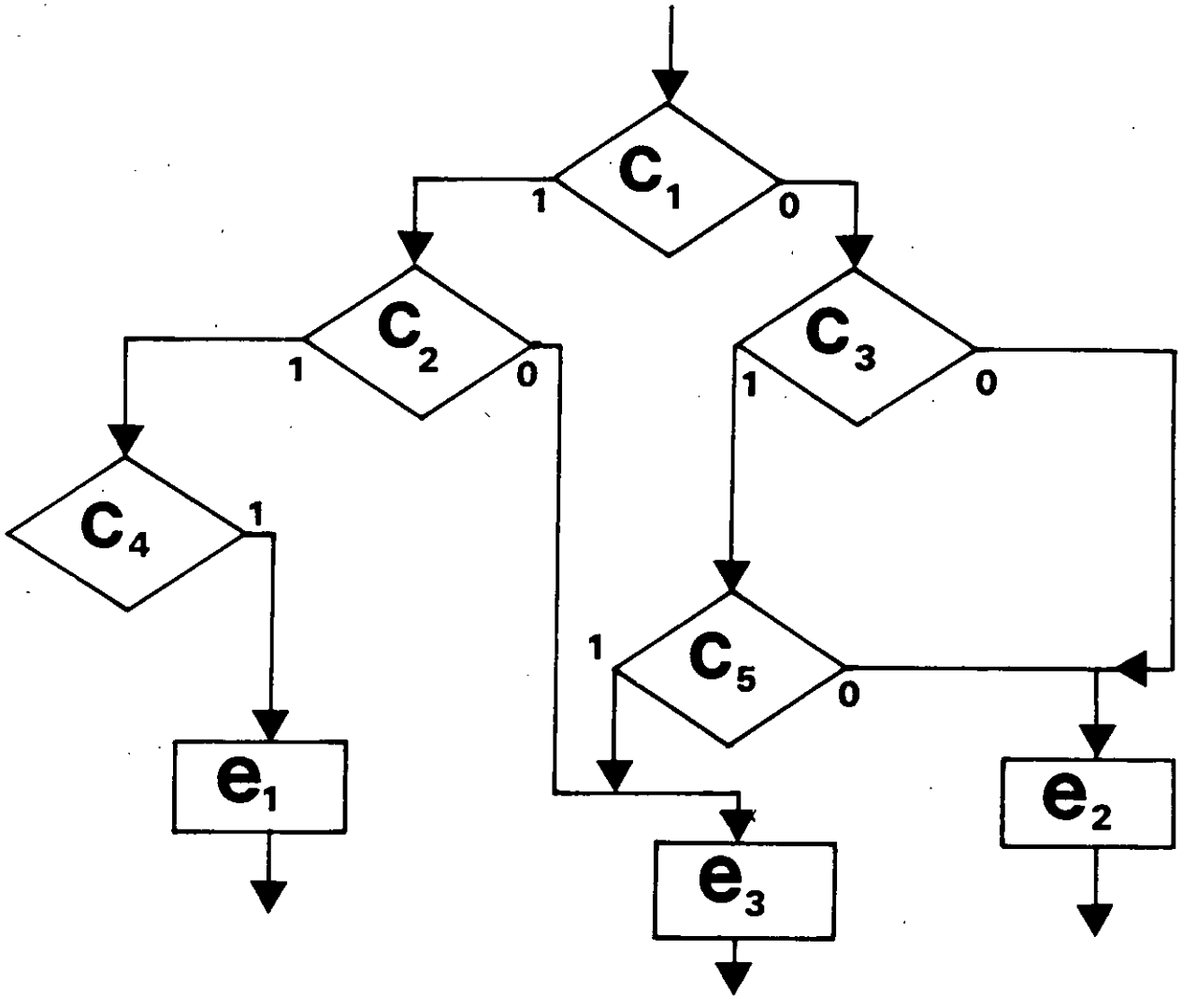
Aprovecharemos lo ya dicho para discutir otra desventaja de los diagramas de flujo que, a nuestro juicio, los vuelve inadecuados para la programación estructurada. Si se observan las figuras de la selección simple y de la iteración condicional simple, es posible creer, erróneamente, que la segunda no es primitiva porque, en apariencia, puede construirse por medio de la primera.

Sin embargo, no es posible construir la iteración condicional a partir de la selección sin tener que recurrir a la flecha que conecta el enunciado de la iteración con su condición, *y esta flecha no está definida como válida en las reglas de formación*.

El equivalente simbólico de la flecha se llama *go to* (ir a) y está, en términos generales, excluido de la programación estructurada debido a su poder "des-

estructurante" y caótico sobre los programas (aunque si se usa con cuidado puede llegar a ser de alguna utilidad*).

En efecto, considérese este nuevo diagrama de flujo, donde hay tantas flechas caóticas que la figura resultante ya no tiene estructura definible, y tiene una sola entrada pero varias salidas. Este hecho vuelve imposible interconectar subfiguras análogas, porque entonces ya no habría un criterio único para tal acción, puesto que se rompió la correspondencia una salida-una entrada. El resultado es un programa que no está estructurado, y que no puede ser analizado ni mantenido fácilmente.



* Suele considerarse a la proposición *go to* como no deseable e incluso como peligrosa, y esto no se debe tan sólo a las razones aquí mencionadas, sino a una cierta mística que acompaña a la programación estructurada, y cuyos orígenes pueden encontrarse —entre otras fuentes— en una carta que el conocido autor Esdger Dijkstra envió a la prestigiada revista *Communications of the ACM* y que apareció publicada en el número de marzo de 1968 con el sugestivo título (puesto por el editor) "La proposición *go to* se considera dañina".

En el siguiente capítulo se emplearán estos nuevos conceptos estructurados para construir una metodología de programación, y se ligarán a las estructuras de datos sencillas, para obtener programas completos, que cumplan su objetivo y sean claros y legibles.

Recomendamos muy especialmente al lector que haga la mayor cantidad posible de ejercicios de programación formal. Es decir, que escriba programas, como los que se han hecho aquí, bien formados, sin preocuparse de para qué puedan servir, o qué problema van a resolver. Este aspecto semántico ("para qué quiero un programa") será analizado más adelante. Por ahora hay que tener la seguridad de que se han entendido cabalmente los conceptos enunciados, y de que se es capaz de manejar con eficacia el anidamiento de estructuras de control, guiados por las reglas recursivas de formación.

Hay que recordar que la base conceptual y matemática de la programación de computadoras indica que los programas pueden ser vistos (si así se deseara) como "teoremas" de un cálculo formal no interpretado, o como cadenas bien formadas de símbolos definidos por medio de reglas libres de ambigüedad. Teniendo esto en cuenta el lector puede dedicarse durante un tiempo a producir programas bien estructurados, haciendo caso omiso por lo pronto de su función semántica.

6.4 Anexo: lenguajes de programación*

En la sección 4.5 se estudiaron los compiladores; aquí se mencionan algunos de los lenguajes de programación más importantes y usuales.

Ada, que debe su nombre a Ada Lovelace, asistente de Babbage en el desarrollo de la máquina analítica, es un intento más por tener un único lenguaje de programación que sea de uso verdaderamente universal. El diseño de este lenguaje, que tomó varios años, fue auspiciado por el departamento de defensa de los Estados Unidos, que exige, desde 1981, que toda la programación que se desarrolle internamente esté escrita en Ada; el grupo que desarrolló Ada estuvo a cargo de Jean Ichbiah. Definitivamente se trata de un lenguaje de características avanzadas, que incluye manejo dinámico de memoria y recursividad, así como facilidades integradas para manejo de programación concurrente, pero la realidad es que no ha tenido la difusión prevista, y es sólo uno más en la gran familia de lenguajes de programación. Más aun, Ada ha causado una gran polémica en términos académicos, porque se arguye que el lenguaje (y su correspondiente compilador) es demasiado grande y poco elegante, y que la tendencia moderna en lenguajes de programación es que sean pequeños y funcionales.

* Consúltense los libros [BARN86], [SAMJ69] y [TUCA87] para una referencia amplia sobre muchos lenguajes de programación, así como el artículo [TESL84]. Véase también el anexo 7.8, que contiene ejemplos de diversos programas escritos en los lenguajes BASIC, C, COBOL, FOREST, FORTRAN y Pascal.

Palabras y conceptos clave

En esta sección se agrupan las palabras y conceptos de importancia que se estudiaron en el capítulo, con el objetivo de que el lector pueda hacer una autoevaluación que consiste en ser capaz de describir con cierta precisión lo que cada término significa, y no sentirse satisfecho hasta haberlo logrado. Se muestran en el orden en que se describieron o se mencionaron.

PROGRAMACIÓN
CODIFICACIÓN
PSEUDOCÓDIGO
ANÁLISIS DE SISTEMAS
ESTRUCTURAS DE CONTROL
ESTRUCTURAS DE DATOS
ENUNCIADOS
SELECCIÓN
DIAGRAMA DE FLUJO
INSTRUCCIONES
ITERACIÓN CONDICIONAL
REGLA DE COMPOSICIÓN
SECUENCIACIÓN
PALABRA CLAVE
PROGRAMACIÓN ESTRUCTURADA

Ejercicios

1. Escriba cinco programas en pseudocódigo y tradúzcalos a sus diagramas de flujo equivalentes.
2. Haga cinco diagramas de flujo estructurados y tradúzcalos a pseudocódigo.
3. ¿Se puede traducir un diagrama de flujo cualquiera a pseudocódigo? ¿Por qué?
4. Los enunciados (e_1, ..., e_n) y las condiciones (C_1, ..., C_n) que se han empleado en los programas en pseudocódigo de este capítulo han sido puramente formales, es decir, carecen de contenido porque no representan nada. Tome cualquiera de los programas ya existentes y sustituya sus enunciados y condiciones por aseveraciones válidas en algún contexto, para obtener un programa con significado y bien formado.

   Por ejemplo, con los enunciados "encender el horno", "esperar un minuto", "mezclar los ingredientes" (aunque es evidente que éste se debe descomponer en varios más), y las condiciones "¿la temperatura es de 300 grados?" y "¿ya está cocido?" se puede describir en pseudocódigo el proceso de cocinar un pastel, desde encender el horno y esperar a que se caliente, hasta que esté terminado.
5. Describa con detalle en pseudocódigo varios procesos usuales de la vida diaria, por ejemplo, entrar en una tienda y comprar un producto (lo cual implica determinar si lo tienen, si el precio es el correcto, formarse en la cola de la caja, etc.), o llegar a un destino en un automóvil o en transporte colectivo.

Referencias para el capítulo 6

[BARN86] Baron, Naomi, *Computer Languages: A Guide for the Perplexed*, Anchor Press/Doubleday, Nueva York, 1986.
Buen libro, con subtítulo místico, que explica las principales características de 22 diferentes lenguajes y los clasifica y califica de acuerdo con su filosofía de diseño y sus facilidades de uso, expresión y legibilidad. Dedica casi 100 páginas a explicar conceptos generales y definir especificaciones estructurales en un nivel introductorio.

[DAHO72] Dahl, Ole; Edsger Dijkstra y Charles Hoare, *Structured Programming*, Academic Press, Nueva York, 1972.
Libro compuesto de tres secciones. En la primera, que lleva el título de "Notes on Structured Programming", el autor holandés Edsger W. Dijkstra, pionero en la teoría tanto de la programación como de los sistemas operativos, menciona los problemas relacionados con "nuestra incapacidad para hacer mucho", y para entender y probar la corrección de los programas. Propone varios esquemas para diagramas de flujo y da un ejemplo completo del sistema para generar programas completos. Es un artículo un poco oscuro (tal vez debido a lo que Dijkstra llama "mis fricciones con el idioma inglés") y hasta un poco rebuscado.

[LINR79] Linger, Richard; Harlan Mills y Bernard Witt, *Structured Programming, Theory and Practice*, Addison-Wesley, Massachusetts, 1979.
Libro de alto nivel matemático sobre la teoría de la programación estructurada, en términos de teoremas y esquemas formales. Propone técnicas de descomposición de programas en diagramas primitivos, e incluye un capítulo sobre pruebas de corrección de programas bien formados. No se trata de un libro para aprender a programar, sino de una referencia teórica.

[MCGC75] McGowan, Clement y John Kelly, *Top-Down Structured Programming Techniques*, Petrocelli/Charter, Nueva York, 1975.
Buen libro sobre programación estructurada que explica las estructuras fundamentales de control; trabaja sobre ejemplos cada vez más complicados hasta llegar a programas recursivos y de computación en paralelo escritos en PL/I. El capítulo cinco explica una idea muy interesante sobre trabajo con equipos de varios programadores que tuvo cierta popularidad, llamada *Chief Programmer Team*.

[SAMJ69] Sammet, Jean, *Programming Languages: History and Fundamentals*, Prentice-Hall, New Jersey, 1969.

La autora escribió un volumen donde explica la historia y aspectos generales de más de cien lenguajes de programación. Aunque es un libro escrito hace varios años, sigue siendo una fuente valiosa y muy reconocida.

[TESL84] Tesler, Lawrence, "Programming Languages", en *Scientific American*, septiembre, 1984.
En este artículo se muestran varios ejemplos de pequeños programas escritos en diversos lenguajes de programación, resaltando los de lenguajes poco usuales, como FORTH, Logo, APL, LISP y Prolog. Es una buena introducción mínima a los lenguajes de programación, que incluye la siguiente observación: "en cierto sentido, la investigación sobre los lenguajes de programación desde 1957 ha estado motivada por intentos de corregir las fallas de FORTRAN".
Existe traducción al español.

[TREJ82] Tremblay, Jean-Paul y Richard Bunt, *Introducción a la ciencia de las computadoras; enfoque algorítmico*, McGraw-Hill, México, 1982.
Traducción de un buen libro sobre computación, original de dos autores canadienses. Está orientado hacia la solución de problemas, y propone un gran número de algoritmos, descritos en pseudocódigo aunque, en nuestra opinión, falta definirlo más formalmente. Trata con bastante amplitud conceptos sobre cadenas de caracteres y estructuras de datos.

[TUCA87] Tucker, Allen, *Lenguajes de programación*, McGraw-Hill, México, 1987.
Traducción de la segunda edición de un texto de lenguajes de programación comparados. Describe con amplitud once diferentes lenguajes, desde FORTRAN hasta Prolog, y tiene además varios capítulos de consideraciones teóricas sobre sintaxis y semántica.

[WIRN84] Wirth, Niklaus, "Data Structures and Algorithms", en *Scientific American*, septiembre, 1984.
Artículo en el que este reconocido autor explora algunas relaciones entre conceptos de estructuras de datos y algoritmos, y muestra varios ejemplos "no triviales" en los que se mencionan los temas de complejidad, corrección y eficiencia de los programas.
Existe traducción al español.

7

Programación estructurada

7.1 Introducción

Este nuevo capítulo comienza con la transcripción de algunas ideas acerca de la programación que se han esbozado en otra parte, y que servirán tanto de resumen de lo ya explicado, como de guía para lo que ahora se comenzará a estudiar.

"Cuando decimos 'programación moderna y estructurada' nos referimos a aquella disciplina que considera el hecho de escribir programas para computadora como un intento serio de aplicar ciertos criterios metodológicos básicos para resolver un problema concreto.

El problema concreto es, por supuesto, escribir un programa que haga lo que se desea resolver con la computadora. Los principios metodológicos básicos son los de subdividir el problema dado en partes asequibles para su análisis, y hacer esto de forma tal que se agilice el proceso de entender por completo tanto el problema como su solución.

Tratándose de programación, estos módulos o subdivisiones deseadas han de cumplir los siguientes requisitos:

A) Deberán estar jerarquizados.
B) Deberán ser pequeños y sencillos.
C) Deberán 'esconder' los detalles poco importantes a módulos superiores en la jerarquía.
D) Deberán, a su vez, usar tantos módulos de más baja jerarquía como sea necesario para cumplir con el punto B).
E) Deberán usar las estructuras de datos y control adecuadas para cumplir con el punto B).
F) Deberán ser legibles; esto es, que no sólo su autor sea capaz de entenderlos, sino cualquiera que tenga acceso a ellos y a un conocimiento elemental de programación.

El punto A) se conoce en la literatura como 'programación de arriba hacia abajo' y se refiere a que un programa consta, en el caso general, de un módulo principal y de varios módulos de nivel más bajo (gobernados por éste) que se encargan de ejecutar las 'órdenes' dadas por él.

El proceso que se emplea para pasar de la descripción de un problema a la generación de un programa estructurado para resolverlo consta de los siguientes pasos:

1. Se propone una solución global al problema en términos de una descripción en un lenguaje llamado pseudocódigo. Esto será el primer acercamiento a la solución. Este pseudocódigo describirá, de manera aproximada, el procedimiento para resolver el problema.
2. Se toma el módulo recién generado y se comienza a refinar progresivamente, tratando de traducir cada una de sus 'pseudoinstrucciones' a órdenes inteligibles para la computadora y verificando, a la par, que esté correcto.
3. Se ejecuta el paso anterior sobre cada uno de los módulos obtenidos, hasta que no quede nada escrito en pseudocódigo y todo haya sido traducido a lenguaje de computadora.
4. Fin.'' (Véase [LEVG80]).

La tarea que nos espera, entonces, tiene varias partes. La primera consiste en ser capaces de describir nuestras ideas acerca de cómo resolver un problema en un lenguaje claro, estricto y universal. El requisito de claridad se cumplirá cuando el programa sea legible e inteligible, y esto va necesariamente ligado a la estructuración del mismo. Que el programa sea estricto quiere decir que deberá estar construido exclusivamente con base en las estructuras ya mencionadas. Por último, que el programa sea universal significa que se podrá transmitir a cualquier interesado en computación y éste lo entenderá por estar escrito en una especie de lenguaje común de programación: el pseudocódigo.

Las otras partes son más complejas; se refieren a la planeación completa de sistemas de programación, en los que intervienen muchos otros factores además de la simple codificación.

Desde hace tiempo se habla de la programación y el diseño estructurados como un factor fundamental en la solución de la llamada ''crisis del software''. Esta situación, común para todo aquel involucrado de alguna manera operativa en la computación, se evidencia cuando los proyectos de programación comienzan a tener retrasos, y cuando se vuelve necesario integrar más gente al grupo de programadores, en un vano esfuerzo por terminar a tiempo.

Es más, se habla incluso de que las medidas tradicionales de productividad tales como año-hombre y mes-hombre son inoperantes tratándose de proyectos de programación y, en general, de proyectos complejos donde intervienen grupos de personal especializado. En efecto, si un albañil construye una pared de un metro de altura en un día, no siempre será el caso que dos de ellos la construyan en medio, porque por lo menos en una ocasión puede suceder

que choquen entre sí y tiren el material que lleven cargando o, incluso, que se estorben de maneras más sutiles, retrasando la entrega del proyecto. Un estudio interesante sobre esto, en el caso específico de la computación, se encuentra en [BROF75].

La referencia [WEIG71], por otro lado, es un interesantísimo estudio de cómo influyen las características de la personalidad en la empresa colectiva de la creación de software y programación, que recomendamos sin reserva a todo aquel interesado en explorar los motivos, tanto técnicos como humanos, de los éxitos y fracasos de este campo de acción.

Esta crisis es más clara cuanto más bajan, por un lado, los precios de los equipos electrónicos, y cuanto más aumentan, por el otro, los costos de los proyectos y los salarios de los analistas y programadores. Considérese tan sólo que en los últimos años los precios de los circuitos integrados se han reducido en magnitud diez veces o más, mientras que los salarios aumentan constantemente.

Comienza, pues, el ejercicio en la primera de las tareas mencionadas.

7.2 Creación de programas en pseudocódigo

Se describieron ya las tres estructuras fundamentales de control, y se sabe también que es posible construir programas válidos sintácticamente anidando en forma recursiva las estructuras entre sí. Se aplicará este conocimiento no sólo para el "cómo", sino también para el "para qué'; es decir, se aprenderá a escribir programas para resolver pequeños problemas prácticos, comenzando con el siguiente: De un conjunto de números, determinar la existencia de alguno en particular; si está en el conjunto, entonces informar dónde se encontró; si no está, informar que no se halló.

Existen varias maneras de resolver problemas de este tipo, llamados de búsqueda. Los **algoritmos de búsqueda** forman parte central de la computación, y tienen gran cantidad de soluciones, unas más eficientes o rápidas que otras. Existen, por ejemplo, algoritmos de "búsqueda lineal", "búsqueda binaria", "búsqueda indizada" o por llave, "árboles de búsqueda", *hashing* o método de dispersiones, etc. La referencia estándar sobre estos temas y, en general, sobre estructuras de datos, sigue siendo [KNUD73].

Se dará solución al problema anterior por medio del algoritmo más sencillo, el de búsqueda lineal.

La idea de este método es simple: se compara el primer número de la lista con el valor buscado; si es igual, se termina la operación con éxito, pero si no lo es, se avanza al siguiente valor. Este proceso continúa hasta que se encuentra el número buscado o se termina la lista. El autor de la referencia recién mencionada, Donald Knuth, propone una versión "mejorada" de este algoritmo tan sencillo, aunque aquí se trabajará sobre la que acabamos de explicar por ser aun más clara.

Una vez comprendido el problema que se desea resolver, el siguiente paso consiste en analizarlo con detalle, tanto en relación con los datos que manejará como de los que se espera produzca como resultados. Para el ejemplo, el

análisis del problema consiste en determinar que es necesaria una lista de números enteros (que por simplicidad se supone dada de antemano), un valor por ser encontrado que se pedirá al usuario, y unos mensajes de terminación con éxito o fracaso. Para el primer caso también hay que indicar la dirección donde se encontró el número pedido, es decir, su posición respecto al inicio de la lista.

Aunque el sistema que se está tomando como ejemplo es casi trivial, se hará su análisis completo, para dar una idea clara del camino a seguir. De lo que se trata es de hacer explícitos los diversos módulos funcionales que integran el problema, separando, por ejemplo, los siguientes: a) un determinado esquema de representación de los datos, b) una función de comparación y decisión para efectuar la búsqueda, y c) una manera de mostrar los resultados.

Ahora hay que averiguar las relaciones estructurales entre estos tres módulos funcionales, o sea, integrar un modelo armónico de funcionamiento del problema con base en las partes mencionadas. Está claro que la función rectora será la búsqueda, que se apoyará en las otras dos. Más aun, el módulo que hace la búsqueda puede, a su vez, subdividirse de manera equivalente, hablando entonces de una función de comparación, una de decisión y otra de repetición.

Llega el momento de hacer una descripción de las interrelaciones entre estas funciones, y es cuando se deben hacer explícitas por medio del pseudocódigo. Se propone la siguiente:

```
! PRIMERA DESCRIPCIÓN DEL PROBLEMA DE BÚSQUEDA
! LINEAL
! Recuérdese que los renglones que comienzan con "!" son comentarios.
! Se supone que existe ya una lista de números enteros
! y un valor para ser localizado en ella.

Observar el primer número de la lista ! esto es, considerarlo el actual

mientras no se termine la lista)
   si el número actual es igual al valor buscado
      entonces informar éxito y terminar
      otro observar el siguiente número  ! esto es, considerarlo el actual
informar fracaso
fin.
```

Para poder seguir adelante, el lector debe comprender perfectamente el funcionamiento del programa recién propuesto. Obsérvese que consiste, en realidad, en la secuenciación de varias proposiciones, una de las cuales es una iteración condicional. Este **mientras** tiene como proposición asociada una selección entre dos posibles acciones.

Es muy importante notar que la única manera de salir del **mientras** es cuando, durante la ejecución, se termine la lista sin haber encontrado el número buscado, por lo que se informa fracaso; es decir, si se hubiera encontrado el número, se habría informado éxito y terminado ahí mismo.

Esto quiere decir que el programa tiene, en realidad, dos salidas: una si hubo éxito (informar éxito y terminar), y la otra luego de haber informado fracaso.

El programa funciona (haga el lector experimentos mentales tomándolo como guía para comprobarlo), aunque se podría hacer un esfuerzo por convertirlo en uno que tenga una sola entrada y una sola salida.

Antes de intentar esto, sin embargo, conviene reflexionar sobre lo que se ha hecho. Se tiene ya un modelo de solución para el problema planteado, que puede ser probado para determinar si es correcto o no. Existen varios tipos de pruebas de programas, pero en el nivel en que se está ahora trabajando nos conformaremos con una prueba mental, que consistirá en examinar el flujo de acciones propuesto por el programa, a la vez que se hacen razonamientos sencillos sobre el mismo. Un ejemplo de este proceder fue la determinación (que no depende de circunstancias) de que el mensaje de fracaso se dará sólo cuando la iteración condicional termine, hecho que únicamente sucederá cuando se agote la lista de números.

Esta facilidad de "avanzar sobre seguro" en la programación es tal vez la ventaja más importante del método propuesto, ya que ahora, durante el diseño, es posible hacer los cambios necesarios para asegurar el funcionamiento correcto del modelo.

Si ya no hay duda acerca de la primera versión, procedemos a modificarla para hacer que tenga una sola salida. Se hará esto sin alterar las relaciones estructurales entre los módulos de búsqueda y expresión de resultados, porque fueron producto del análisis previo.

Lo que se desea hacer es salir del programa en un mismo lugar —al final—, por lo que en lugar de terminar luego de haber informado éxito se levantará una bandera de éxito para dirigirse entonces hacia la salida. Allí habrá un "árbitro" que observará la bandera y decidirá si debe informar éxito o fracaso. Es decir:

```
! SEGUNDA DESCRIPCIÓN DEL PROBLEMA DE BÚSQUEDA
! LINEAL
! Se supone que existe ya una lista de números enteros
! y un valor para ser localizado en ella.

Observar el primer número de la lista  ! esto es, considerarlo el actual

mientras (no se termine la lista)
   si el número actual es igual al valor buscado
      entonces "levantar la bandera de éxito"
      otro observar el siguiente número ! esto es, considerarlo el actual
si la bandera de éxito está levantada entonces informar éxito
                                       otro informar fracaso
fin.
```

Ahora se tiene ya un programa con una sola entrada y una sola salida.

Supóngase que la lista contiene el número que se busca. El programa lo encontrará y levantará la bandera de éxito, pero continuará la iteración (esto

es, no abandona el mientras en ese momento), aunque ya no tenga sentido hacerlo. Aparentemente esto no causa más inconveniente que la pérdida de tiempo; pero puede dar lugar a un problema más sutil. Si se supone que la lista contiene dos o más apariciones del número buscado, el programa encontrará la última de ellas, no la primera. Esto puede estar bien o mal, dependiendo de las especificaciones originales del problema.

Se hará una tercera modificación al programa para evitar la búsqueda redundante en la lista cuando ya se ha localizado el número deseado. (Esta modificación, además, obligará al programa a encontrar la primera aparición del dato pedido, no la última.) Para lograrlo se integrará la pregunta sobre la bandera a la condición del mientras.

```
! TERCERA DESCRIPCIÓN DEL PROBLEMA DE BÚSQUEDA
! LINEAL
! Se supone que existe ya una lista de números enteros
! y un valor para ser localizado en ella.
! Se supone también que inicialmente no se ha encontrado
! lo que se busca.
Bajar la bandera de éxito.
Observar el primer número de la lista ! esto es, considerarlo el actual
mientras (no se termine la lista y la bandera esté abajo)
   si el número actual es igual al valor buscado
      entonces levantar la bandera de éxito
      otro observar el siguiente número  ! esto es, considerarlo el actual
si la bandera de éxito está levantada entonces informar éxito
                                      otro informar fracaso
fin.
```

Esta nueva versión es un poco más compleja, porque ahora la condición que gobierna la iteración es compuesta. Esto obliga a hablar sobre los conocimientos de lógica requeridos para entenderla claramente*.

Para analizar condiciones compuestas como la anterior hay que tomar en cuenta un resultado elemental de lógica, que indica que el valor de verdad (booleano) de la conjunción de dos enunciados es verdadero solamente cuando sus dos componentes son verdaderos, y es falso para las demás combinaciones. Esto puede representarse por medio de la siguiente **tabla de verdad**, cuya primera entrada se lee: "la **conjunción** de un enunciado falso con otro falso tiene valor de verdad falso".

* Se dedicó el anexo 5.4 a mostrar una visión histórica de la ciencia de la lógica matemática; ahí se menciona el origen de varios de los conceptos que se explican ahora. El anexo 7.7 se dedica a una sección de la lógica que se conoce como cálculo proposicional, y que se emplea en los programas de este capítulo y los siguientes.

e1 Λ e2 (e1 y e2)

F	F	F
F	V	F
V	F	F
V	V	V

En forma similar, la tabla de verdad para la **disyunción** es:

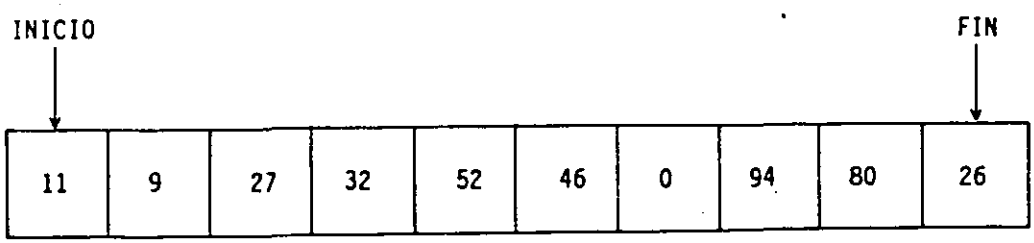
e1 V e2 (e1 o e2)

F	F	F
F	V	V
V	F	V
V	V	V

Para dar un ejemplo, supongamos que son las seis de la tarde de un día muy frío; la conjunción (verdadera) de dos enunciados sería: "hace frío y son las seis", en tanto que "hace frío y son las ocho" es un enunciado compuesto con valor de verdad falso, porque su segunda parte no es cierta. Por otro lado, "hace frío o son las ocho" sigue siendo verdadero aunque no sean todavía las ocho de la noche, si se sigue considerando que la primera parte es verdadera. En este contexto, la frase "hace calor o son las ocho" es falsa porque es la disyunción de dos enunciados falsos.

El lector interesado puede consultar cualquier libro de lógica matemática elemental (y el anexo 7.7) para el estudio de estos temas, que no pueden analizarse aquí con más detenimiento. Una excelente referencia es [TARA68], aunque existen muchas más, como [LOGI70] y [SUPP66].

Ejecutemos el programa sobre la siguiente lista de números:



Para el caso de que se desee buscar el número 27:

1. Bajar la bandera de éxito.
2. Observar el primer número (es decir, el 11).
3. Se entra en el **mientras** porque se cumple que la lista no se ha terminado (tan es así que se está observando su primer dato), y la bandera está abajo (la conjunción de dos enunciados es verdadera si ambos lo son).
4. Como el número 11 no es igual al 27 que se busca, se ejecuta la parte **otro** de la selección, que pide observar el siguiente número (9). Hasta aquí llega el alcance sintáctico de la proposición **mientras**, por lo que el control regresa a evaluar la condición booleana.
5. Se vuelve a entrar al **mientras**, porque los dos enunciados que lo controlan siguen siendo verdaderos.
6. Como el número 9 no es igual al 27 que se busca, se ejecuta la parte **otro** de la selección, que pide observar el siguiente número (27). Hasta aquí llega el alcance sintáctico de la proposición **mientras**, por lo que el control regresa a evaluar la condición booleana.
7. Se vuelve a entrar al **mientras**, porque los dos enunciados que lo controlan siguen siendo verdaderos.
8. Como el número 27 de la lista sí es igual al 27 buscado, se ejecuta la parte **entonces**, y se levanta la bandera de éxito. Hasta aquí llega el alcance sintáctico de la proposición **mientras**, por lo que el control regresa a evaluar la condición booleana.
9. Ahora la evaluación de la condición booleana arroja un valor de verdad falso (porque si bien es cierto que la lista aún no se agota, ya no es cierto que la bandera esté abajo: se acaba de levantar, y la conjunción de un enunciado verdadero con uno falso es falsa). Por tanto ya no se entra en el **mientras**, sino que se abandona para seguir con la ejecución de la siguiente proposición en la secuencia.
10. Esta siguiente proposición es un **si**. Como el valor de verdad de su condición booleana es verdadero, se escoge la parte **entonces**, que indica informar éxito, pues se ha encontrado el valor pedido.
11. Fin.

Aunque ésta no es una demostración de que el programa funciona para todos los casos (pues no se han probado todos; sería imposible por tratarse de un "programa universal" de búsqueda lineal), se invita al lector a convencerse de que sí lo es, por medio de más ejemplos, sobre todo cuando el número buscado esté al inicio o al final de la lista (los llamados "casos límites") o cuando no aparezca en ella (para probar el fracaso). Una segunda forma de comprobarlo es haciendo una abstracción sobre las características lógicas del programa para llegar a la conclusión de que tiene que ser correcto. Esta segunda forma de razonamientos lógicos sobre las proposiciones (que se comenzó a hacer atrás) se vuelve mucho más compleja para el caso de programas largos y sofisticados.

Más adelante se ven formas de convertir esta tercera versión del programa en una función o módulo que pueda ser llamado desde otro lugar, para que

entonces pueda utilizarse múltiples veces. En su estado actual, únicamente ejecuta una vez y luego termina. Esto no parece muy lógico porque, en general, un programa no debería estar diseñado para una sola ejecución, sino para ser ejecutado cuantas veces se requiera.

Una manera sencilla de lograr esto es rodear al programa con un comienza y un termina, incluir todo "dentro" de un

mientras (se desee buscar más datos),

y añadir una proposición para averiguar si se desea seguir o no.

Refinamientos progresivos*

La tercera versión del algoritmo de búsqueda lineal es esencialmente correcta, mas no es ejecutable aún por una computadora. Para que esto sea posible hace falta traducirlo a un programa escrito en un lenguaje de programación, que luego será compilado, ensamblado, cargado y ejecutado de la forma que ya se ha descrito.

La idea fundamental de esta traducción es convertir la "carga semántica" del programa en instrucciones que la ejecuten. Con ese término queremos sintetizar nuestro "conocimiento acerca del mundo", que se empleó para escribir el programa en pseudocódigo y que ahora será necesario bajar de nivel gnoseológico (véase la pág. 126). Esto puede parecer innecesariamente complicado, pero creemos que aquí reside la importancia y potencialidad de la computación. En efecto, ¿de qué serviría una computadora si no fuera más que para remedar, a gran velocidad, algunas capacidades elementales de cálculo? Si hasta aquí llegara su poder, no sería más que una supercalculadora, sin mayor interés adicional. Pero no, se desea hacer consciente al lector de que una computadora puede lograr cosas mucho más complejas que éstas; y es aquí cuando hay que entender el concepto de la "carga semántica" de un programa. Como seres humanos tenemos la posibilidad de conocer el mundo (o aspectos de él) en varios niveles de abstracción. Una máquina, por el contrario, sólo puede "conocer" el mundo en el nivel más elemental de la escala, el instrumental u operativo. Es precisamente cuando la descripción de un fenómeno, proceso o problema no puede ser inmediatamente puesta en términos operativos, cuando nos vemos obligados a traducir este "exceso" semántico de la descripción a elementos más simples. Aquí es donde el pseudocódigo cumple su doble papel de servir como intermediario entre nosotros y la máquina, y de ayudar a la descripción conceptual de un problema en términos de estructuras primitivas. El proceso de esta traducción entre pseudocódigo y un lenguaje de programación ocupa la frontera entre programar y codificar. Aunque no se entrará ahora en los detalles específicos de un lenguaje en particular, sí se llevará más adelante la traducción extrayendo más detalles al pseudocódigo, hasta convertirlo en algo parecido a la codifica-

Consideraciones sobre semántica

* El nombre se debe a uno de los más importantes autores dentro de la programación estructurada, Niklaus Wirth. Esta técnica se presenta en un artículo corto que escribió en 1971. [WIRN71], y se utiliza ampliamente en su libro sobre programación. [WIRN76].

ción de un programa. El proceso que se emplea para esto, descrito al inicio del capítulo, recibe el nombre de *refinamientos progresivos* porque consiste precisamente en obtener cada vez más detalles (dentro de la misma estructura ya definida) hasta llegar a un lenguaje de programación específico.

La diferencia entre el pseudocódigo ya planteado y lo que se quiere obtener está determinada por las estructuras de datos; es decir, hay que especificar cuidadosamente la forma que tendrán los datos en el programa para poder codificarlo.

Por ejemplo, en el programa de la búsqueda lineal, ¿qué quiere decir "mientras haya más elementos en la lista", o "levantar la bandera"? Está claro que se debe hacer más explícita la "instrumentalidad" de estos conceptos si esperamos que una máquina los pueda "entender".

Otra manera de ver la función de las estructuras de datos es precisamente la de descomponer la carga semántica de un programa en elementos más cercanos a la naturaleza operativa de una computadora.

Para nuestros fines utilizaremos una estructura de datos que ya ha sido mencionada antes: el vector (véase la pág. 31). Incluso, el dibujo de la lista de valores de la pág. 219 es precisamente un ejemplo de vector. La manera de declarar este vector como el conjunto de los diez números enteros ahí representados es

```
entero LISTA(10)
```

donde LISTA es el nombre simbólico que se escogió para el arreglo. LISTA(1) entonces representa el primero de esos valores, en tanto que LISTA(*n*) es el enésimo (claro que el valor de *n* debe forzosamente estar entre uno y diez para que esa referencia tenga sentido).

La bandera no será otra cosa que un número entero; cuando su valor sea cero, se dirá que está abajo, y cuando sea uno, que está arriba. ¿Se da cuenta el lector cómo se representa la carga semántica por medio de estas estructuras de datos sencillas?

De la misma forma, "observar un número" de la lista querrá decir apuntarlo por medio del *índice* del vector (el número *n* del ejemplo recién citado). Es por medio de este índice que se traducirá el pseudocódigo de

```
mientras (no se termine la lista y...)
```

a la expresión

```
mientras (el índice no rebase al límite superior y...).
```

Esto es, se utilizará un apuntador para marcar el inicio de la lista y otro para indicar el final. El índice será entonces un número entero que variará entre su valor inicial (comienzo de la lista) y su valor final (terminación de la lista). Estas variaciones (de uno en uno) son la traducción del pseudocódigo "observa el siguiente número".

Tomando todo esto en cuenta se tiene ahora la

```
! CUARTA DESCRIPCION DEL PROBLEMA DE BUSQUEDA
! LINEAL
! Declaraciones de las estructuras de datos usadas.
entero LISTA(10)        ! se supone que ya tiene los valores asignados
                        ! en la figura de la pág. 219.
entero band, ap         ! ap será el apuntador a los elementos de LISTA.
entero INICIO, FIN      ! estos son los límites inferior y superior del vector.
entero VALOR            ! este será el número a buscar dentro de la lista.
! Proposiciones del programa
INICIO = 1; FIN = 10
lee VALOR
band = 0                ! bajar la bandera de éxito
ap = INICIO             ! prepararse para observar el primer número de la lista
mientras (ap <= FIN y band = 0)
   si LISTA(ap) = VALOR
   entonces band = 1    ! levantar la bandera de éxito.
   otro ap = ap + 1     ! alistarse para observar el siguiente número.
si band = 1 entonces
      escribe "Encontré el número", VALOR, "en la posición", ap
      otro escribe "No encontré el número", VALOR
fin.
```

Es muy importante que se comparen atentamente las versiones tercera y cuarta del programa, para verificar que no se han introducido nuevos elementos estructurales, sino que sólo se han limado las "asperezas semánticas", que han sido convertidas en referencias a estructuras de datos sencillas en virtud del cuarto refinamiento.

Los detalles nuevos son los siguientes: ciertas variables numéricas que se han denotado con mayúsculas, que por ahora se suponen con valores dados de antemano (en el caso de la LISTA); variables numéricas denotadas con minúsculas y que actúan dentro del programa, cambiando sus valores según el progreso de la ejecución; expresiones de asignación como

```
band = 1
ap = ap + 1
```

y condiciones booleanas como

```
ap <= FIN y band = 0
LISTA(ap) = VALOR
```

El lector no debe tener problema en asimilar estos detalles, con la posible excepción de las expresiones de asignación, donde el símbolo = no significa "es igual a", sino más bien "se hace igual a", o sea, "se convierte en", perdiendo de esta forma su connotación usual del álgebra, donde sí indica igualdad.

Esto es, la expresión

```
ap = ap + 1
```

no es un sinsentido, puesto que dice "asignar a la variable ap el valor que tenía antes más un uno", o sea, sumar uno a lo que ap valía antes.

Por otro lado, dentro de una condición booleana, el signo = no quiere decir "es igual a" ni tampoco "se convierte en", sino que se lee como la pregunta "¿tiene el mismo valor?". De esta manera, la condición dentro de

```
si LISTA(ap) = VALOR entonces
```

debe leerse "comparar el valor del elemento *ap*-ésimo de LISTA contra el valor de la variable VALOR".

Esto puede resultar un poco confuso a primera vista, porque el signo de igualdad tiene tres usos distintos:

Diversos usos del símbolo =

- Símbolo que denota identidad, como se emplea en aritmética y en álgebra, pero no en programación.
- Símbolo de asignación, como se usa normalmente en lenguajes.
- Símbolo de comparación por igualdad, como se emplea dentro de las condiciones booleanas de los lenguajes de programación.

Para evitar esta ambigüedad, algunos lenguajes de programación (como Algol y Pascal) usan el símbolo doble := para denotar asignación, y el símbolo sencillo = para la prueba de igualdad, pero esto tiene la desventaja de que obliga al usuario a escribir dos caracteres para la asignación y uno solo para la prueba, cuando en la práctica se emplea mucho más la primera que la segunda. En el lenguaje C, más lógicamente, se pide al programador emplear un solo símbolo (=) para el uso más común (asignación), y dos (==) para la prueba de igualdad. Como en realidad los dos usos diferentes del signo de igualdad no pueden confundirse en el contexto de un programa, el lenguaje PL/I lo usa tanto para la asignación como para la prueba, y de la misma forma se hace en el pseudocódigo que se emplea en este libro.

Por último, la expresión

```
escribe "No encontré el número", VALOR
```

manda a la pantalla de la terminal el mensaje que aparece entre comillas, seguido de su valor en ese momento de la ejecución, que es la variable que está luego de la coma.

Sin embargo, lo que se ganó en detalles y estructuras de datos se perdió en la claridad que tenía la segunda versión. Esto es inevitable, puesto que en realidad se está ahora en un proceso tendiente a hacer claro para la máquina lo que antes era claro para nosotros. En virtud de la enorme distancia gnoseológica entre una computadora y el ser humano, cuantos más detalles explícitos tenga un programa menor claridad tendrá, en general, para nosotros.

Esta brecha puede acortarse con buenos hábitos de estructuración en los programas y con una adecuada documentación.

Ya que se tiene esta versión funcionando se puede (y se debe) preguntar si es posible mejorarla. Un análisis más cuidadoso, que parte del ya efectuado, indica que se puede eliminar la bandera si se integra la comparación en el mientras, para que entonces el ciclo principal quede así:

```
! Iteración condicional principal, versión abreviada
ap = INICIO      ! alistarse para observar el primer número de la lista
mientras (LISTA(ap) <> VALOR y ap <= FIN ) ap = ap + 1
si LISTA(ap) = VALOR   entonces info.    ..ito
                       otro informar fracaso
```

Esto es, la iteración condicional terminará cuando se encuentre el VALOR, o bien cuando se termine la LISTA, porque se trata de una conjunción, que se volverá falsa cuando alguno de sus componentes se vuelva falso (cf. pág. 219). El símbolo doble <> quiere decir "diferente de" (es decir, lo contrario de la prueba de igualdad).

Al salir del mientras se pregunta qué fue lo que pasó; si se encontró el valor, se da el mensaje de éxito; si no fue así, quiere decir que el mientras terminó de recorrer el arreglo sin encontrar lo buscado y, por tanto, se da el mensaje de fracaso.

En el anexo 7.8 se muestran las codificaciones reales de la cuarta versión del programa en los lenguajes BASIC, C, COBOL y FOREST; el capítulo 8 trata con detalle la codificación de éste y otros programas en FORTRAN y Pascal. Todos los programas fueron compilados y ejecutados en una computadora. En el mismo anexo se muestra también la versión abreviada recién explicada, codificada en el lenguaje FOREST.

Para el diseño de programas más complejos, sin embargo, es conveniente emplear además nuevas estructuras de control (que ya no serán primitivas, sino compuestas), así como técnicas de programación y diseño más complejas que las estudiadas hasta ahora.

7.3 Estructuras adicionales de control

No obstante que con la secuenciación, la selección y la iteración condicional se puede programar cualquier algoritmo, a veces es más práctico hacer uso de otras estructuras de control, que expresan más directamente lo que se desea hacer. Se mostrarán ahora algunos ejemplos de ellas, junto con un programa que las emplea.

Las nuevas estructuras de control del tipo de iteración condicional son dos: repite y ejecuta.

Ambas son variaciones sobre el mientras, con la diferencia de que primero ejecutan la (o las) proposición asociada y luego evalúan la condición booleana para determinar si se repite el ciclo o no. Recuerde el lector que en el caso

de la iteración condicional original primero se evalúa la condición y después se decide si se ejecuta la proposición.

La sintaxis de la primera versión modificada es:

```
repite
  e
hasta (condición)
```

donde e es un enunciado o proposición.

Aquí dice: "ejecutar repetidamente el enunciado e hasta que la condición se vuelva verdadera", o bien, "ejecutar repetidamente el enunciado e mientras la condición aún no se cumpla".

Se puede demostrar que esta estructura no es primitiva componiéndola como la secuenciación de un enunciado simple y una iteración condicional, para obtener este programa equivalente:

```
e
mientras (no sea la condición) e
```

que primero ejecuta una vez el enunciado y luego averigua si debe o no continuar haciéndolo.

La segunda versión difiere de la primera en cuanto a la forma como se controla la iteración, que ahora está determinada por consideraciones numéricas (el valor de un índice de control), de la siguiente manera:

```
ejecuta i = Li, Ls
      e
```

Este pseudocódigo dice "ejecutar el enunciado e el número de veces indicado por los valores Li (límite inferior) y Ls (límite superior) y dejar de ejecutar cuando el valor del índice sea mayor que el de Ls". Se supone que el índice de control i aumenta de uno en uno. La primera vez se hará igual a Li, la segunda a Li + 1, etc., hasta llegar a ser igual a Ls. Esto marca la última iteración, ya que, al intentar ejecutar la siguiente, el valor del índice será mayor que el límite superior.

El lector podrá comprobar que ese pseudocódigo es la abreviatura de este otro:

```
i = Li
mientras (i <= Ls)
comienza
      e
      i = i + 1
termina
```

con lo que queda claro que no es una estructura de control primitiva o fundamental.

Por el lado de la selección existe una ampliación, llamada caso, que es la abreviatura de

```
si V = A1 entonces e1
          otro si V = A2 entonces e2
                        otro si V = A3 entonces e3
                                      otro ...
```

En efecto, esta construcción anidada no es más que una prueba de opción múltiple sobre el valor de la variable V , y también puede representarse con esta estructura compuesta:

```
caso V de
   A1: e1
   A2: e2
   A3: e3
fin-caso
```

Es importante notar que a lo sumo se ejecutará un solo enunciado, y ningún otro. Incluso puede suceder que no haya ejecución alguna, si la variable V no es igual a alguna de las etiquetas A1, A2, A3.

Para este caso podría decidirse ejecutar un enunciado o proposición que actuará como "sumidero", por el que se desechará el flujo de control, o valor por omisión. Se representará esto añadiendo otro renglón al final, que tendrá una etiqueta vacía como, por ejemplo, en

```
caso ALFA de
     1.6 : tasa = factor * 2
    < 38 : comienza
              lee beta
              escribe zeta1
              tasa = factor * 3
           termina
           tasa = 0
fin-caso
```

La variable **tasa** adquiere un valor específico si sucede que ALFA es igual a 1.6. Si ALFA es mayor que 38, se ejecutan las tres proposiciones indicadas (lee, escribe, nuevo cálculo de **tasa**). Si no se cumpliera ninguno de los dos casos anteriores, entonces se ejecutaría la proposición sumidero **tasa = 0** *.

* Estas nuevas estructuras pueden o no existir en los lenguajes de programación en los que se vayan a escribir los programas. Está claro que cuanto más rico sea el lenguaje escogido, más fácil será pasar de la programación en pseudocódigo a la codificación final. Por ejemplo, no existen en FORTRAN, pero sí (aunque no exactamente) en Pascal, como se verá en el próximo capítulo.

Un ejemplo

Se emplearán ahora algunas de estas nuevas adquisiciones para el diseño de un programa en pseudocódigo para operar con matrices.

Se trata de multiplicar dos matrices y dejar el resultado en una tercera, problema que tiene una especificación clara:

dadas las matrices A, de dimensión $m \times n$
y B, de dimensión $n \times p$,
obtener la matriz producto C, de dimensión $m \times p$.

O, en términos algebraicos:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} * B_{k,j} \qquad \text{para } i = 1, \ldots, m \text{ y } j = 1, \ldots, p$$

Esto quiere decir lo siguiente: "cada elemento de la matriz C (por ejemplo aquel que está en la intersección del renglón i con la columna j) se calculará como la suma de las multiplicaciones (de elemento a elemento) del renglón i-ésimo de A por la columna j-ésima de B".

Por ejemplo, si se tienen las matrices

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad \text{y} \quad B = \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 1 & 2 \end{bmatrix}$$

el resultado de A × B será:

$$\begin{aligned}
(1 \times 7) + (2 \times 9) + (3 \times 1) &= 7 + 18 + 3 = 28 = C_{1,1} \\
(1 \times 8) + (2 \times 10) + (3 \times 2) &= 8 + 20 + 6 = 34 = C_{1,2} \\
(4 \times 7) + (5 \times 9) + (6 \times 1) &= 28 + 45 + 6 = 79 = C_{2,1} \\
(4 \times 8) + (5 \times 10) + (6 \times 2) &= 32 + 50 + 12 = 94 = C_{2,2}
\end{aligned}$$

O sea que la matriz producto es

$$C = \begin{bmatrix} 28 & 34 \\ 79 & 94 \end{bmatrix}$$

Así, se trata de obtener la suma de los productos de cada renglón de A por cada columna de B , para obtener la matriz C.

Dicho con más detalle: es necesario recorrer todos los renglones (denotados por el índice *i*) de la matriz A, e irlos multiplicando por cada elemento de la columna actual de B.

Se propone el siguiente pseudocódigo:

```
! Primera versión de la multiplicación de matrices
! Se suponen las estructuras de datos ya declaradas.
ejecuta i = 1, m
  Hacer las sumas de los productos de los elementos del renglón i-ésimo
  de la matriz A por los elementos de cada columna de la matriz B y co-
  locarlas en el renglón i de la matriz C.
```

Para pasar inmediatamente a este refinamiento:

```
! Segunda versión de la multiplicación de matrices
! Se suponen las estructuras de datos ya declaradas.
ejecuta i = 1, m
   ejecuta j = 1, p
  Hacer la suma de los productos de los elementos del renglón i-ésimo
  de A por los elementos de la columna j-ésima de B y colocarlos en el
  elemento que está en el renglón i y columna j de la matriz C.
```

Ahora hay dos ejecuta anidados, y tal vez valga la pena explicar con un poco de detenimiento qué se obtiene con esto durante la ejecución. El segundo ejecuta está, en efecto, atrapado dentro del alcance sintáctico del primero, por lo que tendrá que terminar todas sus iteraciones (controladas por el índice *j*) antes de regresar a incrementar en uno el índice *i* de la primera construcción iterativa. O sea, se comportan de manera equivalente al odómetro (cuentakilómetros) de un automóvil, donde hay varios engranes anidados de tal manera que por cada ciclo completo de la rueda de los kilómetros, la de las decenas se mueve una unidad; por cada vuelta completa de ésta, la de las centenas se mueve una unidad, y así sucesivamente.

Compruebe el lector la veracidad de lo explicado con el pseudocódigo de esta segunda versión, para darse cuenta de que se hará la suma de los productos de cada renglón de A (el *i*-ésimo en cada caso) por los valores de todas las *p* columnas de B (de una en una, pues están controladas por el índice *j* que, variando de igual forma, llega desde la primera hasta la última).

Ahora hay que especificar con más detalle cuál es ese elemento de la matriz A y cuál el de B:

```
! Tercer refinamiento de la multiplicación de matrices
! Se suponen las estructuras de datos ya declaradas.
ejecuta i = 1, m
   ejecuta j = 1, p
      ejecuta k = 1, n
```

```
Hacer la suma de los productos del elemento (i, k) de la matriz A
por el elemento (k, j) de la matriz B y colocarla como el elemento
(i, j) de la matriz C.
```

Por último, es necesario asegurarse de que cada elemento de la matriz C tendrá ceros antes de ser usado, para no alterar los resultados finales:

```
! Cuarto refinamiento de la multiplicación de matrices
entero A(m,n), B(n,p), C(m,p)
ejecuta i = 1, m
   ejecuta j = 1, p
   comienza
      C(i, j) = 0
      ejecuta k = i, n
         C(i, j) = C(i, j) + ( A(i, k) * B(k, j) )
   termina
```

En el anexo 7.8 se muestra una versión de este programa, codificada en CBASIC.

Se vuelve a formular al lector la petición que se le hiciera antes: practique cuanto pueda.

7.4 Módulos y subrutinas

Podría decirse que la regla general de la burocracia (y de la mala política) es: "cuando no puedas decidir algo, delega la responsabilidad a alguien más". Ahora, pondremos a trabajar este principio, para buenos fines, en programación, y se mencionarán las técnicas más importantes para lograrlo.

En un sistema de programación no trivial, generalmente se ven involucradas múltiples funciones por desempeñar en tiempos que no siempre son definibles de antemano, sino que cambian dependiendo de la ejecución y las características de los datos que procesa el programa. Si se intentara incluir en el cuerpo mismo del programa todas las funciones que un determinado sistema tiene que cumplir, se llegaría a un resultado oscuro y difícil de entender. Por otro lado, todo sistema incluye tareas rutinarias y poco importantes que tienen que efectuarse, y cuya descripción pormenorizada simplemente estorbaría dentro del programa. Tales aspectos obligan a delegar estas funciones (difíciles para el primer caso y rutinarias para el segundo) a **módulos** que realizarán precisamente esas tareas.

Concepto de módulo

Se podría definir un módulo diciendo que es una unidad autocontenida de código (o sea, dice todo lo que se requiere para entenderlo) y que tiene la característica de no ser directamente ejecutable, sino que debe ser llamado para efectuar sus funciones. Un módulo tiene atribuciones para poder llamar

a otro si fuera necesario, por lo que la delegación de responsabilidades es una propiedad general. Existe, por fuerza, un módulo principal que es el que invoca a los demás, siendo él mismo directamente ejecutable; se le conoce como módulo o programa principal.

Los buenos hábitos de programación indican que el módulo principal debe ser pequeño y fácil de entender, y estar constituido casi por completo de llamadas a otros módulos de más bajo nivel que realicen efectivamente el trabajo. Desde este punto de vista, la función del programa principal será coordinar a los otros módulos. En la siguiente sección se explicará con detalle este tipo de organización. Por lo pronto hay que aprender a definir y llamar módulos en pseudocódigo, quedando entendido que los detalles variarán dependiendo del lenguaje de programación que se escoja para la codificación.

Un módulo comenzará con la palabra clave proc (abreviatura de procedimiento) seguida del **nombre simbólico** que se haya escogido para él. A continuación viene el **cuerpo** del módulo, que consta de las declaraciones y proposiciones necesarias para efectuar su función (igual que en el caso de los programas). En los puntos de terminación lógica del módulo se escribe la palabra clave regresa. Puede haber varios de estos puntos dentro de un módulo, que dependen del orden de la ejecución. Se marca el final de proc (punto de terminación física) con la palabra fin, al igual que se hizo en el caso de los programas.

Operación de los módulos

Por otro lado, se logra llamar a un módulo escribiendo la palabra clave llama seguida del nombre simbólico del módulo deseado.

Entonces, el esquema completo funciona así: se definen los módulos necesarios por un lado, y se llaman dentro del programa principal cuando sean requeridos. Cuando esto sucede, el flujo de control pasa automáticamente al cuerpo del procedimiento llamado. La ejecución continúa dentro de este módulo hasta que se ejecuta alguna proposición regresa, luego de la cual el flujo retorna al módulo que mandó llamar al que recién termina. De aquí en adelante la ejecución continuará con el flujo normal, hasta que se encuentre con otra llamada (si es que existe). Como se dijo antes, no solamente el programa principal puede llamar módulos, sino que éstos pueden llamar a otros, tantos como sea necesario.

Lo siguiente es un ejemplo de la definición de un módulo, y de su llamada por el programa principal:

```
proc principal
   declaraciones
   declaraciones
   declaraciones
   proposición 1
   proposición 2
   proposición 3
   proposición 4
   llama uno
   proposición 5
   proposición 6
fin.
```

```
proc uno
   declaraciones...
   declaraciones...
   proposición a
   proposición b
   proposición c
   regresa
fin.
```

Y éste es el orden de ejecución del programa principal:

```
proposición 1
proposición 2
proposición 3
proposición 4
proposición a
proposición b
proposición c
proposición 5
proposición 6
```

Sucederá muchas veces que no sólo se esté interesado en requerir un módulo determinado, sino que además se desee darle ciertos datos para que los procese. De la misma manera, muchas veces un módulo tomará cierto valor, lo modificará y lo devolverá al módulo que lo mandó llamar. Un ejemplo sencillo de esto podría ser un módulo que recibe dos valores, los suma y devuelve el resultado.

Estas variables, que sirven para pasar valores y datos entre módulos, reciben el nombre de parámetros. Existen dos tipos de parámetros: reales, y formales o fingidos. Los primeros contienen valores que el módulo principal pasa al que está siendo llamado, mientras que los segundos son simplemente los nombres locales con que los reconoce el módulo que fue llamado.

En el siguiente ejemplo de un programa principal y un módulo (o rutina) se observa cómo se plantea el paso de parámetros:

```
proc principal                      proc suma (s1,s2,r)
entero a, b, resultado              entero s1, s2, r
lee a, b                            r = s1 + s2
llama suma (a,b, resultado)         regresa
escribe "a + b =", resultado        fin.
fin.
```

Aunque está claro que nadie pensaría en emplear tantos recursos para efectuar una simple suma, se aprovechará lo ya escrito para identificar las variables enteras **a**, **b** y **resultado** como los parámetros reales, y **s1**, **s2** y **r** como los parámetros formales, que simplemente ocupan el espacio para recibir (al tiempo de la llamada) los valores reales mandados por el procedimiento principal.

Siempre deberá cumplirse que la declaración de los parámetros reales coincida con sus correspondientes parámetros formales. En este ejemplo todas son variables enteras.

Otro nombre que suele darse a los parámetros reales es el de argumentos.

Por supuesto que ahora sería posible decir, dentro del programa principal o dentro de cualquier otro módulo,

```
llama suma (m,n,z),
```

y el mismo procedimiento de suma funcionará, esta vez con argumentos diferentes de los de la primera llamada.

Esta ya es una ventaja de los módulos o subrutinas: permiten ahorrar código, pues basta con llamarlos tantas veces como se desee para que vuelvan a funcionar cuando fueron, por supuesto, definidos una sola vez.

Se tendría un ejemplo más realista si se estuviera programando un sistema en el que hubiera que hacer búsquedas en listas diferentes y desde lugares diferentes. Bastaría entonces con convertir el programa busca en una subrutina o módulo que pudiera llamarse por cualquier sección del sistema que requiriera una búsqueda con esas características. El encabezado de esta rutina sería, por ejemplo:

```
proc busca (LISTA, n, INICIO, FIN, VALOR, band, ap)
```

El cuerpo de este módulo sería esencialmente igual al definido en la sección anterior, con excepción de que los valores que antes se suponían como dados de antemano ahora se pasan como argumentos.

Dicha función se llamaría de esta manera:

```
llama busca (ALFA, 50, 7, 14, -21, band, ap).
```

El pedido de búsqueda dice: "en el arreglo llamado ALFA, que tiene 50 posiciones, buscar la aparición del número –21 entre las posiciones 7 y 14; si se encuentra ese valor, devolver la variable band con un uno y ap con la dirección donde estuvo; en otro caso, band y ap valdrán cero".

Pero también podría ser llamado de esta otra manera:

```
llama busca (BETA, 10, 2, 9, "A" , band, ap),
```

pidiendo buscar la aparición de la letra A entre la segunda y la novena posiciones del arreglo BETA, que tiene diez*.

Tal vez el lector ya se dio cuenta de otra enorme ventaja de los módulos, enfocada ya no a ahorrar renglones del programa, sino más bien a organizar funcionalmente las acciones dentro de un sistema de programación por medio de su capacidad sintetizadora, que permite definir y asignar funciones de acuerdo con un plan maestro sin tener que resolver cada vez localmente problemas que pueden ser atendidos en un solo lugar.

Es más, esta característica de los módulos da lugar a toda una metodología de programación y diseño de sistemas que se explica enseguida.

---

* Hay que tomar en cuenta que esto está en pseudocódigo. Será más o menos difícil codificar esto en un lenguaje específico de programación, dependiendo de las facilidades modulares que tenga.

7.5 Técnicas de diseño descendente

Las ideas expuestas en relación con la inesperada capacidad organizativa de los módulos son tan importantes que sirven como base para un conjunto de métodos y esquemas globales para construir sistemas completos de programación, que recibe el nombre de **diseño estructurado**

Existen varias "escuelas" de diseño estructurado, entre las que se pueden mencionar las de Jackson, [JACM75], Yourdon, [DEMT79], y Warnier-Orr, [ORRK77]. Aquí se estudiarán maneras sencillas de lograr los mismos resultados, y la guía seguirá siendo más intuitiva que formal.

Quizá lo más importante de tales técnicas sea la necesidad de tener un plan global de acción que diga, en principio, cómo va a funcionar el sistema, haciendo caso omiso de detalles y funciones poco importantes o no estructurales. La primera tarea consiste en escribir una versión inicial del módulo principal, que indique claramente la forma como se va a repartir el trabajo entre los demás módulos.

Esto no tiene en realidad nada de novedoso, y es prácticamente la única manera de atacar cualquier problema complejo. Así funciona cualquier plan de gobierno de un país, por ejemplo, y es también la forma en que están coordinados los organismos ejecutivos nacionales. Es decir, existe un plan maestro (que funge como rector) y hay tantos delegados (ministros) como sean necesarios para atender, en un primer nivel, las tareas y funciones por éste especificadas. El director de cada uno de los primeros niveles está capacitado para repetir a su vez el esquema por el cual él mismo fue creado, o sea, para elegir tantos cuadros de segundo nivel como su función lo indique. Esta delegación de responsabilidades continúa hasta los niveles operativos propiamente dichos, que son los que efectúan el trabajo operativo sobre los elementos de la realidad.

Todo este esquema organizativo se vendría abajo si no estuviera asegurado que cada módulo va a cumplir la misión encomendada por el que lo creó (como por desgracia suele ocurrir en la realidad política*).

En un sistema de programación es relativamente fácil cumplir con este requisito, mediante algo parecido a las banderas que ya hemos utilizado, y por medio también de la facilidad de "modularizar" el sistema tanto como sea necesario.

Un aspecto muy importante de todo esto es el siguiente: *no es necesario esperar a que un módulo esté totalmente terminado para incluirlo en el plan de acción de otro superior en la jerarquía.*

Ventajas de la jerarquía de módulos

Lo anterior equivale a decir que se puede terminar de programar un módulo antes de finalizar con los que éste va a llamar. Esto, a su vez, significa que es posible tener un modelo global funcionando aun antes de que estén codificados (o incluso programados) todos los módulos de un sistema.

Para lograrlo, se introduce el concepto de módulo nulo o vacío (llamado *stub* en la literatura computacional en inglés), que simplemente ocupa el lugar que más tarde se llenará con código ejecutable. Lo siguiente es un ejem-

* "La política es infinitamente más complicada que la física" dijo Albert Einstein cuando rechazó la presidencia honoraria del estado de Israel, que le fuera ofrecida a comienzos de la década de 1950.

plo de un módulo vacío que tendrá que cumplir más adelante la función especializada de encontrar un registro dentro de un archivo e imprimir su valor:

```
proc localiza (archl, n, result)
     escribe "localiza: supongo que ya encontré el registro número", n,
             "dentro del archivo", archl
     llama imprime (registro)
regresa
fin.
```

Y el módulo de impresión "vacío" será

```
proc imprime (reg)
     escribe "imprime: supongo que ya imprimí el registro pedido"
regresa
fin.
```

El programa principal podrá ser ejecutado e invocará a este módulo tantas veces como sea necesario dentro de su plan de acción. Claro que la ejecución no procesará ningún dato real, pero sí le indicará al analista o al programador que todo marcha de acuerdo con el esquema planeado originalmente. Si el módulo principal dice, por ejemplo . . .

```
proc principal
   .
   .
   .
   lee clase
   .
   .
   si clase = clientes entonces llama localiza (clientes, clave, result)
                       otro...
   .
   .
   .
  fin.
```

ya se puede tener una primera versión del sistema completo —estructuralmente—, bastante tiempo antes de su terminación, lo que a su vez permite tener una idea de si las cosas marchan o no.

Esta metodología de diseño recibe el nombre de "descendente" porque parte de los módulos de más alto nivel, delegando responsabilidades a los de más abajo cuando así convenga, y sin tener que esperar a que éstos estén terminados.

Un resumen de los pasos a seguir es:

1. Escribir el módulo principal, que tendrá las siguientes funciones:
   —Controlar el flujo global del programa o sistema.
   —Definir la jerarquización de las funciones llamando a nuevos módulos esclavos.

2. Definir en forma descendente cada uno de los módulos a que se haya hecho referencia anteriormente. Cada uno de ellos podrá comportarse como si fuera el subprincipal, que controla todo un subsistema que depende de él. Todo módulo deberá cumplir una sola función (o hacer una sola cosa) desde su propio punto de vista (*i. e.*, puede ser compleja y requerir de la participación de otros módulos auxiliares, pero sigue siendo, en un primer nivel jerárquico, una sola función).

3. Probar el diseño completo usando los módulos nulos.

4. Refinar progresivamente cada uno de los módulos vacíos, cuyas existencia y posición jerárquica ya han sido determinadas.

En virtud de todo lo ya expuesto, es evidente que los módulos que configuran el sistema están fuertemente interrelacionados, de manera que comparten algunos datos (ya sea globales, o por medio del paso de parámetros) y producen resultados que les son solicitados por otros. Esto no quiere decir, sin embargo, que todos los módulos de un sistema conocen los datos y valores de todos los demás; por el contrario, es imprescindible que la jerarquización controle todos los elementos que manejan los módulos, determinando cuáles de ellos tendrán acceso a cuáles datos, y bajo qué circunstancias. Sería grave, por ejemplo, que en un sistema de programación los módulos "de abajo" pudieran alterar información que se requiere en el primer nivel para el control de toda la operación. De la misma forma, es deseable que las entradas y salidas de datos en un sistema estén agrupadas en unos cuantos módulos especializados, para evitar que cada uno lea y escriba datos sin control.

Estas son algunas de las reglas que gobiernan la convivencia entre módulos en un sistema estructurado:

Reglas para el manejo de los módulos

- Todos los módulos deberán estar jerarquizados.
- Un módulo deberá hacer *una* sola cosa (aunque ésta pueda ser de alto nivel).
- Los módulos de arriba tienen derechos sobre los de abajo (en forma recursiva):
  —Definen las funciones de los que dependen de ellos.
  —Exigen resultados correctos sin preocuparse de detalles ni de la dificultad para obtenerlos.
  —No son llamados por los de más bajo nivel.
- Los módulos de bajo nivel deberán ocultar su funcionamiento y detalles de operación ante los de arriba.

- El intercambio de información entre módulos es de dos tipos:
  —De arriba hacia abajo (descendentemente) pasan órdenes y argumentos.
  —De abajo hacia arriba se devuelven resultados en forma de parámetros.
- La interacción entre módulos debe ser clara y explícita, mediante invocaciones como llama o equivalentes.
- Todos los módulos deberán tener una sola entrada y una sola salida.
- El tamaño deseable para cada módulo es de aproximadamente 50 ó 100 líneas de lenguaje fuente, para que sea comprensible sin gran esfuerzo y sin tener que leer páginas y más páginas. Además, y puesto que cada módulo efectúa una sola función, no será difícil enterarse de lo que cada uno hace.

Escribir un sistema de programación se vuelve una tarea muy diferente de las que hasta entonces se realizaban: ya no es necesario tener todos los programas codificados para probar en la computadora; ya es posible determinar si se va por buen camino en la construcción de una aplicación especial, porque se tienen adelantos de la ejecución (por medio de los módulos nulos) y, además, para diseñar el módulo principal basta con incluir llamadas a nuevas subrutinas cuando sea necesario. Luego llegará el momento de programarlas con detalle; por lo pronto, ya son utilizables, lo que reduce el tiempo total para la terminación del proyecto.

Así pues, estamos listos para un último ejemplo, que consistirá en el diseño y programación de un programa totalmente simbólico, para subrayar también la capacidad de la computación para simular procesos reales (del mundo) y rebatir la idea de que únicamente sirve para calcular.

Un problema simbólico

Se escoge un problema conocido (resuelto ya en la referencia mencionada al comienzo de este capítulo) llamado "problema de las ocho damas", donde hay que encontrar una forma (puede haber varias) de colocar ocho damas de ajedrez en un tablero, de tal manera que no se ataquen entre sí. El gran matemático alemán Carl Friedrich Gauss (1777-1855) intentó buscar una elusiva solución analítica para este pasatiempo. Pruebe el lector en este momento colocarlas de la manera pedida y comprenderá inmediatamente que se trata de un problema no trivial.

Un primer análisis dirá que éste es el máximo número de damas que podrían colocarse de esta manera en un tablero, pues éste sólo tiene ocho columnas (o renglones) y es imposible colocar nueve o más ya que, entonces, por lo menos dos damas quedarían en la misma columna (o renglón).

Nos podríamos preocupar por diseñar un algoritmo que fuera decidiendo dónde colocar cada dama, haciendo consideraciones tal vez muy complejas (por ejemplo, de tipo combinatorio), pero decidimos plantear esta sencilla idea: se comenzará por colocar una dama en la primera casilla de la primera columna, para intentar luego colocar sucesivamente las demás, una en cada columna adicional, observando siempre que no se ataquen.

Pronto llegará el momento en que ya no se pueda colocar una dama determinada en un lugar seguro. Cuando esto suceda, simplemente se volverá a la dama anterior (o sea, a la columna anterior) y se dirá que ese lugar parecía

seguro, pero en realidad impedía colocar las subsecuentes piezas, por lo que se renuncia a él; entonces, se escoge la siguiente casilla permisible dentro de esa columna, si es que la hay. Si no existe, se vuelve a la dama anterior y se repite el proceso. Esto terminará, luego de varios cientos (o miles) de retrocesos, con éxito. Debe ser así, ya que se ha procedido de manera sistemática, y siempre hacia adelante; incluso los retrocesos se hacen de manera ordenada y como un paso para poder seguir adelante.

¿Qué se hace luego de haber encontrado una solución? Se imprime o se informa de alguna manera y nos preparamos para encontrar más, usando el mismo esquema de retrocesos. Se abandona la octava dama (solamente existe una casilla libre para ella: aquella donde acaba de estar), se regresa a la séptima y se intenta colocarla más abajo en su columna; si no es posible, entonces se abandona y se intenta con la sexta, etcétera.

Llegará el momento en que se intente mover la primera dama más allá de su octava casilla, y esto significará que se han encontrado todas las maneras posibles de colocar las ocho damas. No cometeremos la indiscreción de decir cuántas soluciones existen, porque se espera que sea el programa del lector interesado (ya codificado y ejecutando) el que lo averigüe.

Pero estar todavía muy lejos de saberlo con precisión tiene poco que ver con la principal ventaja del método de trabajo, porque ya es posible producir la primera versión en pseudocódigo del programa:

```
! Primera versión del problema de "las ocho damas"
! Se supone que existe un tablero.
colocar la primera dama en la casilla uno de la columna uno
repite
comienza
   mientras (haya más damas por colocar)
   comienza
      escoger la siguiente dama (en orden de columna ascendente)
      tratar de encontrarle un lugar en la columna correspondiente
      si se encuentra entonces colocarla allí
                        otro
                        comienza
                  descartar esa dama
                  volver a la anterior y considerarla como si fuera la nueva
                        termina
   termina
   imprimir la solución encontrada
   prepararse para la siguiente, o sea, abandonar la octava dama y
   considerar a la séptima como si fuera la nueva
termina
hasta (que no haya más soluciones)
```

Es fácil ver que esta primera versión es esencialmente correcta aunque no considera los detalles (o tal vez precisamente por eso); desde este momento podemos comenzar a probarla mentalmente.

Ahora sí se puede y se debe comenzar a preocuparse por los aspectos operativos y por cierta cantidad de detalles adicionales, todo a partir de la primera versión que, aunque sea desde un punto de vista conceptual, ha demostrado ser correcta.

Se necesitan varios módulos de segundo nivel para resolver funciones bien específicas. Uno de ellos se llamará `libre`, y tendrá como parámetro una variable booleana **resultado**, que será verdadera si el módulo fue capaz de encontrar el lugar pedido dentro de la columna actual, y falsa en caso contrario. Además, devolverá otros parámetros que apuntarán a la casilla donde quedó colocada, si tuvo éxito.

Otro módulo se llamará **coloca** y su función será colocar la dama actual en una posición determinada que le será pasada como parámetro (será, por supuesto, el segundo parámetro de `libre`).

Un módulo más se llamará **atrás**, y se encargará de abandonar la dama actual y prepararse para tomar la anterior.

Así, se obtiene la

```
! Segunda versión del problema de "las ocho damas"
! Se supone que existe un tablero.
llama coloca (posición)
repite
comienza
   mientras (la dama actual no sea la octava)
   comienza
      llama libre (posición, resultado)
      si resultado entonces llama coloca (posición)
                    otro llama atrás (posición)
   termina
   imprimir la solución encontrada
   llama atrás (posición)
termina
hasta (que la dama actual sea anterior a la primera)
```

Llegó el momento de preocuparse por los detalles para representar una dama, y averiguar si una cierta posición en el tablero es segura o no. Obsérvese que la hipótesis de trabajo es que los módulos son (o serán) capaces de hacer esto, porque precisamente para ello se crearon y se están llamando; dado que la programación es descendente, ha llegado el momento de resolver cómo.

Se propone la siguiente matriz como la estructura de datos para representar el tablero:

COLUMNAS

RENGLONES

1,1	1,2	1,3	1,4	...	1,8
2,1	2,2	2,3	2,4	...	2,8
.	.			...	.
.		.		...	.
.			.	...	.
8,1	8,2	8,3	8,4	...	8,8

donde cada pareja (**ren, col**) especificará unívocamente una casilla; es decir, cada dama estará asociada con una pareja de éstas cuando esté en el tablero, pues son los parámetros de posición.

Ahora viene la consideración más importante. Se observa que una condición necesaria y suficiente para que dos damas se ataquen es que:

A) estén en un mismo renglón (esto es, tienen igual el primer número del par), o que
B) estén en la misma columna (esto es, tienen igual el segundo número del par), o que
C) estén en la misma diagonal que sube (esto es, el resultado de restar el segundo número del primero es igual para ambas damas), o que
D) estén en la misma diagonal que baja (esto es, la suma de sus dos números de posición es igual).

Se instruirá al módulo **libre** para que antes de intentar colocar la n-ésima dama verifique que ninguna de estas cuatro condiciones se cumpla con las n-1 damas que ya están en el tablero, para n entre 2 y 8.

Éste es el primero de los módulos, parcialmente refinado:

```
proc libre (ren,col,resultado)
! buscar una posición libre para la dama descrita
! devolver resultado = VERDADERO si se encontró
mientras (ren < 8)
comienza
   resultado = VERDADERO
   ren = ren + 1 ! avanzar la dama una casilla dentro de su columna
   repite
      si la diagonal hacia arriba está amenazada o
         la diagonal hacia bajo está amenazada o
```

```
        el renglón actual está amenazado entonces resultado = FALSO
    hasta (terminar de analizar todas las columnas anteriores o
        resultado = FALSO)
    si resultado = VERDADERO entonces regresa ! se encontró una
                                              ! posición libre
termina
regresa
fin.
```

Este es el módulo que coloca las damas en una posición dada:

```
proc coloca (ren,col)
colocar la dama en la casilla ren de la columna col y tomar nota de que
ese renglón del tablero ya quedó amenazado. Tomar nota de que la diago-
nal que sube y que cruza esa casilla, junto con la diagonal que baja,
quedan también amenazadas.
regresa
fin.
```

Éste otro es el módulo para hacer retroceder una dama:

```
proc atrás (ren,col,última)
col = col - 1  ! abandonar la dama actual
! liberar las diagonales que estaban amenazadas por esa dama.
! liberar el renglón que estaba amenazado por esa dama.
si ya no es posible avanzar la primera dama más
                                    entonces última = VERDADERO
regresa
fin.
```

Sólo hace falta un módulo para imprimir los resultados de manera adecuada. La codificación final del programa se muestra en el anexo 7.8, en lenguaje C. El próximo capítulo trata sobre la codificación detallada de éste y otros programas, en los lenguajes FORTRAN y Pascal.

7.6 Documentación y prueba de programas

Si el lector ha seguido con detalle todos los pasos de programación explicados a lo largo del texto y se ha esforzado en hacer ejercicios, como repetidamente se ha solicitado, entonces podemos hablar ya de las últimas etapas en la creación de un programa o sistema de programación completos.

Como se dijo en la introducción, es más importante programar que codificar, lo cual indica que los programas para computadora también son, en gran medida, ejercicios de escritura comparables a una literatura menor.

Esto significa que un programa debe ser claro y entendible no sólo para una máquina, sino también para sus creadores o usuarios.

Los usuarios de un sistema

Es posible distinguir varios tipos de usuarios de un programa: los beneficiados (o afectados) finales, los usuarios directos, y los encargados de su mantenimiento y operación. Está claro que los del primer tipo son la mayoría, y es sobre ellos que recae el impacto de las computadoras en la sociedad. No obstante, ocurre que los sistemas de programación tienen que estar diseñados con cierta "ingeniería humana", y con un mínimo de conocimiento del papel que las computadoras y la informática desempeñan en nuestros entornos sociológicos.

Centraremos la atención en el segundo tipo de usuarios, pues son ellos los que supuestamente se beneficiarán de forma inmediata con el programa o sistema que se haya escrito. Piénsese, por ejemplo, en la situación común en los cursos de circuitos eléctricos de las facultades de ingeniería: llega el momento en que los estudiantes ya no pueden efectuar manualmente los complejos cálculos necesarios para resolver redes de elementos eléctricos, y tienen que recurrir a la computadora (o a las calculadoras programables) para dar solución a casos particulares. Si se decidiera escribir un programa para analizar interactivamente redes pasivas, se podría diseñar un sistema completo, que fuera llamado desde una terminal y auxiliara directamente a cada alumno en su problema específico, interactuando con él y presentándole respuestas inmediatas a los planteamientos propuestos hasta llegar a la mejor solución. Por supuesto, tal usuario potencial no tiene por qué entender cómo se diseñó el programa, ni tiene tampoco por qué entender el código ni todo lo relativo; basta con que sea capaz de usar eficientemente el sistema para poder explotarlo adecuadamente. Esto obliga a escribir un manual para el usuario que sea claro, explícito y completo.

Las características que un manual para el usuario debe tener para ser aceptable son, entre otras, una explicación teórica del problema que resuelve, una explicación somera de cómo lo resuelve, una idea acerca de la filosofía del diseño escogido (por qué éste y no otro, etc.), y una explicación pormenorizada de su uso; y en esto último deberán presentarse ejemplos de la operación real del sistema junto con explicaciones de qué hacer en caso de falla o comportamiento extraño, además de un mínimo de supuestos acerca del conocimiento que el usuario tiene sobre el sistema. Todo programa o grupo de programas deberá siempre permitir al cliente modificar los datos con los que lo alimentó, así como darle la capacidad de interrumpirlo en cualquier momento. Haciendo un símil, siempre se debe evitar la situación enojosa de aquel que va conduciendo, pasa por equivocación la salida que le correspondía en una carretera y tiene que pagar su olvido con un rodeo de muchos kilómetros; hay que dejar —en el mismo sentido figurado— la posibilidad de retornos y salidas a lo largo del camino.

Hablando del tercer tipo de usuarios, los encargados del mantenimiento y operación de los programas, diremos que son los únicos que deben estar enterados en detalle de cómo y por qué funciona el sistema. Deberán, por tanto, tener a mano la documentación técnica adecuada para cumplir con su papel. Existen dos tipos de documentación técnica: la descrita en un **manual de diseño** y la contenida en el programa fuente mismo.

La primera es el equivalente computacional del manual para el usuario, en el sentido de que lleva de la mano al lector, ya no en el uso final del sistema, sino en su funcionamiento interno. Un buen manual debe contener, entre otras cosas, un diagrama que muestre el flujo de la información en el sistema, junto con una descripción de los algoritmos empleados y un conjunto de los esquemas más significativos de pseudocódigo generados por medio de los refinamientos progresivos. También es importante mostrar las principales estructuras de datos usadas en el diseño.

Función estructural de los comentarios en un programa

La documentación contenida en el cuerpo de los programas mismos recibe el nombre genérico de **comentarios**, que incluso se han empleado en los ejemplos. Los comentarios tienen la importantísima función de hacer explícita la "distancia semántica" entre los renglones del programa fuente y los conceptos que el programador tenía en mente cuando los escribió. En este sentido está claro que los comentarios no se limitarán simplemente a remedar el código, pues no tendrían razón de ser; tal es el caso, por ejemplo, del siguiente comentario:

```
ALFA = ALFA + 1  ! Se suma uno a la variable ALFA
```

En cambio, este otro comentario sí tiene utilidad:

```
ALFA = ALFA + 1  ! Se apunta al siguiente elemento libre
```

suponiendo que la variable ALFA tiene la función de servir como índice de elementos de cierto arreglo.

La idea general consiste en convertir las versiones viejas de los refinamientos en comentarios para las nuevas. De esta manera, con seguridad, siempre se estará al día en la documentación del sistema, pues no habrá necesidad de inventar comentarios para meterlos en el programa fuente. Asimismo, la aritmética caprichosa que pide escribir "un comentario por cada cinco renglones de código" carece totalmente de sentido, pues no parte de las consideraciones semánticas que ya se han mencionado.

Esto quiere decir que la metodología de refinamientos progresivos es completa, en el sentido de que respeta la historia del desarrollo intelectual del sistema y permite que aparezca en la documentación. Lo único que hay que hacer es archivar las hojas donde se han efectuado los refinamientos, para poder volver a ellas en caso necesario (para usarlas como documentación o para rastrear errores recién descubiertos).

Además, se deben incluir en cada módulo comentarios que den una explicación de, al menos:

- función del módulo
- datos y parámetros de entrada
- datos y parámetros de salida
- estructuras de datos principales que emplea
- nivel jerárquico
- módulos que lo llaman
- módulos que llama

Documentación mínima para cada módulo

Se sugiere que estos comentarios estén agrupados al inicio de cada módulo, y que sean claramente visibles.

El lector interesado en profundizar más sobre los temas de documentación de programas y sistemas puede consultar la excelente referencia [KERB78].

Prueba de programas

Como se mencionó antes, el concepto de prueba de programas se ha modificado enfocándose cada vez más a corregir detalles operativos que a determinar si el sistema ya codificado cumple o no las funciones para las que fue hecho.

Esto implica que los analistas y programadores están seguros de la funcionalidad de los programas aun antes de terminar con la codificación, por haberlos diseñado de manera estructurada.

Se prevé que el programa omita ciertos detalles o cometa algunos errores cuando se ejecuta por primera vez en la computadora, mas esto no implica que esté mal estructuralmente; es equivalente a descubrir pequeñas grietas en las paredes de un edificio terminado, o darse cuenta de que algunos focos no encienden, o que las llaves de agua del baño gotean, detalles de poca importancia global.

Viene entonces una etapa de corrección de detalles, hasta dejar el producto terminado con las características de elegancia y funcionalidad pactadas.

También puede suceder, sin embargo, que se detecten errores estructurales cuando se ha terminado la codificación de un sistema, en cuyo caso habrá que planear con cuidado las medidas a tomar. Quizá no se haya comprendido bien lo que se deseaba (lo cual es a todas luces catastrófico a estas alturas, y se espera que ya no suceda), o que no se haya hecho correctamente el análisis preliminar. El que ocurra cualquiera de estas dos cosas es grave y habla de una mala planeación en las etapas del desarrollo. El único antídoto contra este mal consiste en una buena dosis de recato y moderación en el momento de la planeación (junto con una sólida experiencia que permita ofrecer proyectos robustos y maduros), por un lado, y un buen entendimiento de la metodología científica de programación y diseño estructurados, por el otro.

En los sistemas realmente grandes (telecomunicaciones, sistemas operativos, etc.), se vuelve virtualmente imposible prever todos los casos de falla no estructural, por lo que se requiere de un apoyo constante de mantenimiento y atención a quejas. Se diseñan incluso formas impresas donde se pide al usuario directo que anote las fallas que detecte, para su posterior corrección.

Además, existen técnicas matemáticas para asegurar la corrección de los programas, y muchos autores proponen que tal debe ser el camino a seguir en el diseño de programas y más aun en los cursos introductorios. [HOAC69] es uno de los artículos que inició con este enfoque, que pide considerar la programación como producción de esquemas sobre los cuales se deben hacer razonamientos formales y demostraciones de consistencia y corrección. En el libro [ULLJ76] (mencionado en el capítulo 4) se dedica un capítulo a estos métodos, que son formalizaciones de los razonamientos que aquí se han iniciado sobre los programas, tales como los de la pág. 218 y en [HARD87] se pueden encontrar también algunos de estos temas, en un nivel introductorio.

[GRID81], por ejemplo, representa una fuente autorizada dedicada de lleno a lo que podría llamarse la ciencia de la programación (a diferencia del carácter de "arte" que se le había dado hasta entonces), y ése es precisamente el título del libro.

Esto ha continuado en los últimos años, y existe ahora una fuerte corriente de pensamiento en la que se trata a los programas como teoremas, y la programación como una tarea formalizable. Hay métodos de diseño basados en metodologías matemáticas bien definidas, entre los que sobresale la conocida como "programación funcional".

Para nuestros fines, sólo resta mostrar al lector la codificación de todos los programas usados en el libro, para que los utilice como referencia y pueda compararlos contra los pseudocódigos mostrados. Si se han aprendido los fundamentos de la programación estructurada ya no será tan ardua la tarea de continuar.

7.7 Anexo: elementos de lógica proposicional

En el capítulo 5 se habló sobre lógica matemática y se dedicó un anexo a la historia de su desarrollo; ahora se hará una breve introducción al tema específico del cálculo (o lógica) proposicional, que es una interpretación del álgebra desarrollada el siglo pasado por George Boole. Para ello se darán incialmente varias definiciones.

Una variable es un término que no posee significado por sí mismo.

Cuando en una expresión que contiene variables se sustituyen éstas por constantes (términos con significado preciso) determinadas, se convierte en una proposición, o función proposicional, que tiene la característica de tener uno y sólo uno de los valores permitidos por el sistema lógico en cuestión. Como se trata de un álgebra con dos elementos constantes, entonces una proposición puede tomar únicamente dos valores: verdadero o falso, pero no ambos a la vez. Es necesario mencionar también (explícita o implícitamente) a los cuantificadores, que son los que convierten una expresión con variables en una proposición.

Si p denota una proposición, entonces hay dos casos posibles:

1) p es verdadera
2) p es falsa

y si p es verdadera entonces no es falsa, y viceversa.

Esos son los **valores de verdad** de la variable p.

En el ejemplo anterior p fue utilizada como variable y también como proposición, de una manera general.

Se define la negación de p, denotada $-p$, como aquella proposición que es verdadera cuando p es falsa, y viceversa.

Claramente, el operador "la negación de" ($-$) es unario; esto es, su rango de acción está limitado a una sola fórmula proposicional (que bien puede ser una variable aislada, como en el ejemplo).

Se empleó ya una notación tabular muy común, llamada **tabla de verdad** para ejemplificar en forma compacta todos los posibles casos en que se utiliza tal o cual operador.

La tabla de verdad para la negación es la siguiente:

p	$-p$
V	F
F	V

Se define la **conjunción** de dos variables o proposiciones, p y q, denotada $p \wedge q$, como aquella que es verdadera cuando ambas, p y q, son verdaderas; y falsa en caso contrario.

La **disyunción** de p y q se denota $p \vee q$, y se define como falsa cuando ambas, p y q, son falsas; y verdadera en caso contrario. Hay que notar que se utiliza la versión completa de la disyunción, es decir, $p \vee q$ significa "p o q, o ambas". Esto se conoce como "*o* inclusivo", que en latín se llama *vel*, de donde proviene el símbolo V.

Las tablas de verdad de los operadores binarios de conjunción y disyunción se presentaron, en la pág. 219.

Desde otro punto de vista, se está hablando de aplicaciones (mapeos) del producto cartesiano $S \times S$ en S, donde S es el conjunto dotado de dos elementos: (V,F) o bien (1,0).

Ahora bien, $S \times S$ tiene cuatro elementos: (V, V), (V, F), (F, V) y (F, F). Habrá entonces $2^4 = 16$ aplicaciones de $S \times S$ en S, porque cada uno de los cuatro elementos de $S \times S$ se aplica a los dos elementos de S, y a cada una de las aplicaciones se asigna uno de los valores, F o V.

Entre las 16 posibles aplicaciones se encuentran las mencionadas hasta ahora, junto con varias otras que son muy útiles; hay algunas, finalmente, que son poco utilizadas.

Existen 10 conectivos binarios que, sumados a las dos variables (en dos casos especiales llamados tautología y contradicción) y a las dos variables negadas, dan el total de 16 funciones.

Una de ellas es la función conocida con el nombre **NOR** (*not or*), también llamada operador de Peirce, en honor del lógico y filósofo estadounidense Charles Sanders Peirce (1839-1914), denotada como $p \mid q$, y que se lee "ni p ni q"; es de particular importancia en los circuitos lógicos.

La función **NAND** (*not and*), también llamada functor de Sheffer, se denota como $p \mid q$, y se lee "no es el caso que p y q"; es de gran importancia práctica también en diseño de circuitos lógicos.

La **implicación material** de p y q, denotada como $p \rightarrow q$, es otra de esas 16 funciones, y se lee "si p entonces q". De ella se derivan varias concepciones, tales como las de condición necesaria y condición suficiente.

Por ejemplo, en la frase "si nieva entonces me da frío", en realidad se está haciendo la implicación de un **antecedente** p ("nieva") y de un **consecuente** q ("me da frío"). Se dice que es condición suficiente que nieve para que me dé frío; pero no es necesario que nieve para tener frío, puedo tener frío por alguna otra causa.

Sin embargo, en la frase "si me da frío es porque nieva" ($q \rightarrow p$), entonces el tener frío es condición necesaria de que nieve.

La unión de las dos declaraciones anteriores resultaría en algo como "si nieva me da frío", y "si me da frío quiere decir que nieva", o más simplemente como "me da frío si, y sólo si, nieva". Se diría entonces que p es condición necesaria y suficiente para q, o $p \leftrightarrow q$. Ésta es la **bicondicional** o **equivalencia**, que se abrevia como "ssi". En condiciones normales la proposición "me da frío ssi nieva" es falsa, porque ya hace frío aun antes de comenzar a nevar, pero la proposición "hoy es martes ssi ayer fue lunes" es siempre cierta.

Esto lleva a dos aplicaciones más que tienen particular trascendencia: la **tautología** y la **contradicción**.

La tautología tiene la particularidad de ser siempre verdadera, sin importar las particularidades o estado lógico de sus componentes, mientras que la contradicción es siempre falsa, para toda combinación de sus componentes.

En la obra del gran lógico Ludwig Wittgenstein, mencionada en el capítulo 5. *Tractatus Logico Philosophicus*, se lee: "(5.143) la contradicción se oculta, por así decirlo, fuera de todas las proposiciones; la tautología, dentro. (4.464) La verdad de la tautología es cierta; la de las proposiciones, posible; la de las contradicciones, imposible".

Es conveniente decir, además, que bastan algunas funciones de todas las posibles para poder definir las restantes. Por ejemplo, es posible definir, a partir de las funciones Λ y $-$, las restantes.

Con el functor de Sheffer únicamente, es posible definir todas las funciones restantes. Igualmente es posible lograr esto sólo con el operador de Peirce.

En la práctica es necesario trabajar y construir funciones de varias variables, donde éstas están relacionadas entre sí mediante cadenas de operadores lógicos.

Surge entonces el problema de saber —dada una fórmula cualquiera— si está construida correctamente, si puede tener algún sentido. Esto ya se trató en el capítulo 5, cuando se abordó el tema de la computabilidad.

El estudio del cálculo proposicional continúa con los diversos métodos para definición y simplificación de funciones compuestas, y sirve como base conceptual para la creación de circuitos electrónicos que se comportan de acuerdo con las tablas de verdad especificadas que, a su vez, se emplean como elementos constructivos en el diseño de las computadoras.

En el anexo 6.4 se mencionó un lenguaje de programación (Prolog) cuya filosofía está basada en el cálculo proposicional, y en el que es relativamente fácil construir y probar expresiones lógicas complejas, cosa que se dificulta un tanto con los demás lenguajes.

7.8 Anexo: ejemplo de programas codificados en diversos lenguajes

Este anexo contiene la codificación del pseudocódigo de la pág. 223 (algoritmo de búsqueda lineal) en los lenguajes de programación BASIC, C, COBOL, FOREST, Forth, FORTRAN, LISP, Modula-2, Pascal y Prolog, como una ilustración de las características de un programa totalmente terminado en estos lenguajes. Es importante mencionar que se presentan sólo como ejemplos visuales de programas ya codificados; no se espera que el lector los pueda seguir con detalle, pues no lo logrará sino hasta el final del estudio de este texto.

Se presenta también la versión final en FOREST de la versión abreviada de la búsqueda, cuyo pseudocódigo aparece en la pág. 225.

Luego está la codificación en CBASIC del programa de la pág. 230 para multiplicar matrices.

Por último se presenta la codificación en lenguaje C del programa de las ocho damas, de las págs. 239-241, junto con un ejemplo de sus resultados.

El lector interesado en la codificación tiene todavía por delante el capítulo 8, que se dedica por completo al diseño de programas codificados en FORTRAN y Pascal, y que incluye varios ejemplos y programas terminados. Además, le recomendamos desde ahora que, una vez que sepa programar en pseudocódigo, se ejercite en el lenguaje de su preferencia con alguno de los múltiples libros que existen para tal efecto ([FRIF84]: FORTRAN, [FRIF86]: BASIC, [GROP86]: Pascal, [KERB85]: C, [PHIA87]: COBOL).

Todos los programas de búsqueda son equivalentes, y producen resultados como el siguiente, que fue copiado directamente de la computadora donde se ejecutó. Aparecen subrayados los datos que se teclearon como respuesta a las peticiones del programa interactivo:

```
La lista es: 11,9,27,32,52,46,0,94,88,26

DAME EL VALOR A BUSCAR: 11
ENCONTRE EL NUMERO  11  EN LA LOCALIDAD No. 1

DESEAS CONTINUAR? (s/n): s
La lista es: 11,9,27,32,52,46,0,94,88,26

DAME EL VALOR A BUSCAR: 26
ENCONTRE EL NUMERO  26  EN LA LOCALIDAD No. 10

DESEAS CONTINUAR? (s/n): s
La lista es: 11,9,27,32,52,46,0,94,88,26

DAME EL VALOR A BUSCAR: 14
EL NUMERO 14 NO ESTA EN LA LISTA

DESEAS CONTINUAR? (s/n): s
La lista es: 11,9,27,32,52,46,0,94,88,26

DAME EL VALOR A BUSCAR: 0
ENCONTRE EL NUMERO  0  EN LA LOCALIDAD No. 7

DESEAS CONTINUAR? (s/n): n
ADIOS.
```

```
rem Codificacion del programa de busqueda lineal en el lenguaje CBASIC
rem Los "comentarios" en BASIC deben tener la palabra "rem"
rem      al inico del renglon.
    dim LISTA%(10)
rem valores iniciales
    data 1,10
    data 11,9,27,32,52,46,0,94,88,26
    data "s"
    read INICIO%
    read FIN%
    for i% = INICIO% to FIN%
      read LISTA%(i%)
    next i%
    read si$
rem  Iteracion condicional principal
    while si$ = "s" or si$ = "S"
         band% = 0
         ap% = INICIO%
         print : print
         print " La lista es: ";
         for i% = INICIO% to FIN%
           print "-"; LISTA%(i%) ;
         next i%
         print : print
         input " DAME EL VALOR A BUSCAR: "; VALOR%
         print VALOR%
         rem iteracion condicional para la busqueda
         while ap% <= FIN% and band% = 0
              if LISTA%(ap%) = VALOR% then band% = 1    \  rem lo encontre.
                                      else ap% = ap% + 1  rem sigo buscando
         wend
         if band% = 1        \
         then print : print " ENCONTRE EL NUMERO "; VALOR%;  \
                            " EN LA LOCALIDAD "; ap%         \
         else print : print " EL NUMERO "; VALOR%; " NO ESTA EN LA LISTA"
         print
         input " DESEAS CONTINUAR? (s/n): "; si$
         print si$
    wend
    print : print " ADIOS. "
    stop
    end
```

```
/* Codificacion del programa de busqueda lineal en el lenguaje C */

/* Valores iniciales  */
int LISTA[10] = {11, 9, 27, 32, 52, 46, 0, 94, 88, 26} ;
int INICIO = 0 ;
int FIN = 9 ;
char si = 's' ;

main()
{
int VALOR, band, ap, i ;

/*  Iteracion condicional principal  */
while( si == 's' ) {
    band = 0 ;   /*  baja la bandera  */
    ap = INICIO ;
    printf( " \n\nLa lista es: " ) ;
    for( i = INICIO ; i < FIN ; i++ ) printf( "%d,", LISTA[i] ) ;
    printf( "%d", LISTA[FIN] ) ;
    printf( " \n\nDAME EL VALOR A SER BUSCADO: ") ;
    scanf ( "%d", &VALOR ) ;
    printf( "%d", VALOR ) ;
/*  Iteracion condicional para la busqueda  */
    while( ap <= FIN && band == 0 )
         if( LISTA[ap] == VALOR )
           band = 1 ;   /*  lo encontre: sube la bandera  */
           else ++ap ;  /*  aun no lo encuentro, continuo buscando  */
    if( band == 1 )
      printf( " \nENCONTRE EL NUMERO  %d  EN LA LOCALIDAD No. %d\n",
                                                    VALOR, ap+1 ) ;
      else printf( "\nEL NUMERO %d NO ESTA EN LA LISTA\n", VALOR ) ;

    printf( "\n¿DESEAS CONTINUAR? (s/n): " ) ;
    scanf ( "%s", &si ) ;
    printf( "%c", si ) ;
}
printf( "\n\nADIOS.\n" ) ;
}
```

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.   BUSCA-COB.
*******************************************
*   CODIFICACION DEL PROGRAMA DE BUSQUEDA  *
*         LINEAL EN EL LENGUAJE COBOL      *
*******************************************
 ENVIRONMENT  DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.    Z-8000.
 OBJECT-COMPUTER.    Z-8000.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01   TABLA.
       02  FILLER    PIC X(29) VALUE
       "11,09,27,32,52,46,00,94,88,26".
 01   LISTAS         REDEFINES TABLA.
       02  TIPOS     OCCURS 10 TIMES.
       03  LISTA     PIC XX.
       03  COMA      PIC X.
 01   INICIO         PIC 99 VALUE 01.
 01   FIN            PIC 99 VALUE 10.
 01   APUNTADOR      PIC 99 VALUE ZEROES.
 01   VALOR          PIC 99 VALUE ZEROES.
 01   OTRA-BUSQUEDA  PIC X  VALUE SPACES.
 01   BANDERA        PIC X  VALUE SPACES.
 PROCEDURE DIVISION.
 EFECTUA-BUSQUEDAS.
      MOVE "S" TO OTRA-BUSQUEDA
      PERFORM EFECTUA-UNA-BUSQUEDA
        UNTIL OTRA-BUSQUEDA = "N".
      STOP RUN.
 EFECTUA-UNA-BUSQUEDA.
      PERFORM DESPLIEGA-LISTA.
      PERFORM ACEPTA-VALOR.
      MOVE 0 TO BANDERA.
      MOVE INICIO TO APUNTADOR.
      PERFORM DETALLE-BUSQUEDA
        UNTIL APUNTADOR > FIN OR
              BANDERA NOT = 0.
      IF BANDERA = 1
        PERFORM REPORTA-EXITO
      ELSE
        PERFORM REPORTA-FRACASO.
      MOVE SPACE TO OTRA-BUSQUEDA.
      PERFORM ACEPTA-OTRA-BUSQUEDA
        UNTIL OTRA-BUSQUEDA = "S" OR
              OTRA-BUSQUEDA = "N".
```

```
 DETALLE-BUSQUEDA.
      IF LISTA (APUNTADOR) = VALOR
        MOVE 1 TO BANDERA
      ELSE
        ADD 1 TO APUNTADOR.
 DESPLIEGA-LISTA.
      DISPLAY "      " LINE 22 ERASE.
      DISPLAY " LA LISTA ES: " LINE 4 POSITION 10
      DISPLAY TABLA LINE 4 POSITION 25.
 ACEPTA-VALOR.
      DISPLAY " DAME EL VALOR A SER BUSCADO: "
              LINE 7 POSITION 10.
      ACCEPT  VALOR LINE 7 POSITION 40.
 REPORTA-FRACASO.
        DISPLAY "EL NUMERO "


       LINE 9 POSITION 15
       DISPLAY VALOR
       LINE 9 POSITION 25
       DISPLAY " NO ESTA EN LA LISTA"
       LINE 9 POSITION 28.
ACEPTA-OTRA-BUSQUEDA.
     DISPLAY "DESEAS CONTINUAR? (S/N):"
             LINE 18 POSITION 10
     ACCEPT OTRA-BUSQUEDA  LINE 18 POSITION 38.
REPORTA-EXITO.
       DISPLAY " ENCONTRE EL NUMERO "
               LINE 11 POSITION 10
       DISPLAY VALOR
               LINE 11 POSITION 32
       DISPLAY "EN LA LOCALIDAD No."
               LINE 11 POSITION 35
       DISPLAY APUNTADOR
               LINE 11 POSITION 55.
```

```
  Codificacion del programa de busqueda lineal en el lenguaje FOREST
! Los "comentarios" en FOREST deben comenzar con un signo de admiracion.
integer LISTA(10)
integer INICIO, FIN, VALOR, ap, band, xi
!
! Valores iniciales
data LISTA / 11, 9, 27, 32, 52, 46, 0, 94, 88, 26 /
data INICIO / 1 /
data FIN / 10 /
data xi / 1Hs /
!
mientras( xi = "s" )
comienza
  band = 0  ;  ap = INICIO
  write(6,100) ( LISTA(i), i = INICIO, FIN )
  100 format( //, " La lista es: ", 50i3 )
  write(6,110) ; 110 format(//," DAME EL VALOR A BUSCAR: ")
  read(5,120) VALOR  ;  120 format(i3)
  write(6,120) VALOR
  ! iteracion condicional para la busqueda
  mientras( ap <= FIN & band = 0 )
     si( LISTA(ap) = VALOR ) band = 1   ! lo encontre, subo la bandera
     otro ++ap                          ! continuo buscando
  si( band = 1 ) write(6,130) VALOR, ap
  otro write(6,140) VALOR
  130 format(//," ENCONTRE EL NUMERO", i3, " EN LA LOCALIDAD No.", i3 )
  140 format(//," EL NUMERO", i3, " NO ESTA EN LA LISTA" )
  write(6,150)   ;  150 format(//, " DESEAS CONTINUAR? (s/n):" )
  read(5,160) xi ;  160 format( 1a1 )
termina  ! fin de la iteracion principal
write(6,170)  ;  170 format(//, " ADIOS." )
end
```

```
( PROGRAMA DE BUSQUEDA LINEAL ESCRITO )
( EN EL LENGUAJE FORTH )
: V VARIABLE ;
V LISTA 18 ALLOT
V BAND
V AP
V INICIO
V FIN
V VALOR
: ASIGNA  1 INICIO ! 10 FIN ! 11 9 27 32 52 46 0 94 88 26
10 0 DO LISTA 9 I - 2 * + ! LOOP ;
ASIGNA
     : BUSCA VALOR ! 0 BAND ! INICIO @ AP !
     BEGIN BAND  @ 0 = AP  @ FIN @ <= AND WHILE
     LISTA AP @ 1 - 2 * + @ VALOR @ = IF
     1 BAND ! ELSE 1 AP + ! THEN REPEAT BAND @ 1 = CR IF
     ." ENCONTRE EL NUMERO " VALOR @ . ." EN LA POSICION "
     AP @ . ELSE ." NO ENCONTRE EL NUMERO " VALOR @ . THEN
     CR CR ;
```

```
C Codificacion del programa de busqueda lineal en el lenguaje FORTRAN
C Los "comentarios" en FORTRAN deben tener la letra "C" en la columna 1.
      integer LISTA(10)
      integer INICIO, FIN, VALOR, ap, band, si

C Valores iniciales
      data LISTA / 11, 9, 27, 32, 52, 46, 0, 94, 88, 26 /
      data INICIO / 1 /
      data FIN / 10 /
      data si / 1Hs /

   10 continue
      if( si .ne. 1Hs ) go to 50
      band = 0
      ap = INICIO
      write(6,100) ( LISTA(i), i = INICIO, FIN )
  100 format( //, " La lista es: ", 50i3 )
      write(6,110)
  110 format(//," DAME EL VALOR A BUSCAR: ")
      read(5,120) VALOR
  120 format(i3)
      write(6,120) VALOR
C simulacion de la iteracion condicional para la busqueda
   20 continue
      if( ap .gt. FIN .or. band .ne. 0 ) go to 30
C lo encontre; subo la bandera
      if( LISTA(ap) .eq. VALOR ) band = 1
C aun no lo encuentro, continuo buscando
      if( LISTA(ap) .ne. VALOR ) ap = ap + 1
      go to 20
   30 continue
      if( band .eq. 1 ) write(6,130) VALOR, ap
  130 format(//," ENCONTRE EL NUMERO", i3, " EN LA LOCALIDAD No.", i3 )
      if( band .ne. 1 ) write(6,140) VALOR
  140 format(//," EL NUMERO", i3, " NO ESTA EN LA LISTA" )
      write(6,150)
  150 format(//, " DESEAS CONTINUAR? (s/n):" )
      read(5,160) si
  160 format( 1a1 )
      go to 10
C simulacion de fin del -mientras-
   50 continue
      write(6,170)
  170 format(//, " ADIOS." )
      end
```

```
; PROGRAMA DE BUSQUEDA LINEAL EN EL LENGUAJE LISP

(SETQ LISTA '(11 9 27 32 52 46 0 94 88 26))

(DEFUN PRINCIPAL (LAMBDA (LISTA)
  (LOOP
    (MUESTRALIS LISTA)
    (PREGVAL)
; SE LEE EL VALOR POR BUSCAR Y SE INICIA EL PROCESO
    (SETQ POS (BUSCA (SETQ DATO (RATOM)) LISTA 1))
    (EVALUA DATO POS)
    (PREGCONT)
    ((EQ 'N (RATOM)) ) ) ))

(DEFUN MUESTRALIS (LAMBDA (LISTA)
  (PRINT '"LA LISTA ES: ")
  (PRINT LISTA) ))

(DEFUN PREGVAL (LAMBDA NIL
  (PRINT '"DAME EL VALOR A BUSCAR ") ))

(DEFUN BUSCA (LAMBDA (ELEMENTO LISTA CONTADOR)
; PROCEDIMIENTO RECURSIVO DE BUSQUEDA
  ((NULL LISTA) LISTA)
  ((EQ ELEMENTO (CAR LISTA)) CONTADOR)
  (BUSCA ELEMENTO (CDR LISTA) (PLUS 1 CONTADOR)) ))

(DEFUN EVALUA (LAMBDA (DATO POS)
  ((NULL POS) (FRACASO DATO) )
  (EXITO DATO POS)
   T ))

(DEFUN FRACASO (LAMBDA (NUMERO)
  (PRINT '"EL NUMERO ")
  (PRINT NUMERO)
  (PRINT '"NO ESTA EN LA LISTA")
  NIL ))

(DEFUN EXITO (LAMBDA (NUMERO POSICION)
  (PRINT '"ENCONTRE EL NUMERO ")
  (PRINT NUMERO)
  (PRINT '"EN LA LOCALIDAD ")
  (PRINT POSICION) ))

(DEFUN PREGCONT (LAMBDA NIL
  (PRINT '"DESEAS CONTINUAR? (S/N): ") ))
```

```
MODULE Busca ;
 (* Programa de busqueda lineal en el lenguaje Modula-2 *)
 (* Se selecciona en primer lugar el modulo de donde se
    importaran los procedimientos de lectura y escritura *)
  FROM InOut IMPORT WriteLn, WriteString, WriteCard, ReadCard, Read ;
  CONST
    Inicio = 1 ;
    Fin = 10 ;
  TYPE
    Rango = [Inicio..Fin] ;
    (* En Modula-2 hay un tipo para los enteros positivos: CARDINAL *)
  VAR
    Lista : ARRAY Rango OF CARDINAL ;
    ap, i, band, VALOR : CARDINAL ;
    mas : CHAR ;
  BEGIN
    (* Al igual que en Pascal, en Modula-2 no existe una manera elegante
       de dar valores iniciales a los arreglos *)
    Lista[1] := 11 ;
    Lista[2] :=  9 ;
    Lista[3] := 27 ;
    Lista[4] := 32 ;
    Lista[5] := 52 ;
    Lista[6] := 46 ;
    Lista[7] :=  0 ;
    Lista[8] := 94 ;
    Lista[9] := 88 ;
    Lista[10] := 26 ;
    mas := 's' ;
    WHILE mas = 's' DO
      band := 0 ;
      ap := Inicio ;
      WriteLn() ;
      WriteLn() ;
      WriteString('La lista es: ') ;
      FOR i := Inicio TO Fin DO
        WriteCard(Lista[i], 4) ;
        WriteLn() ;
```

```
    END ;
    WriteLn() ;
    WriteString('Dame el valor a buscar: ') ;
    ReadCard(VALOR) ;
    WriteCard(VALOR, 4) ;
    (* Iteracion para la busqueda *)
    WHILE (ap <= Fin) AND (band = 0) DO
      IF Lista[ap] = VALOR THEN
        band := 1 ;          (* se encontro *)
      ELSE
        ap := ap + 1 ;
      END ;
    END ;
    IF band = 1 THEN
      WriteLn() ;
      WriteString('Encontre el numero ') ;
      WriteCard(VALOR, 4) ;
      WriteString(' en la localidad No. ' ) ;
      WriteCard(ap, 3) ;
    ELSE
      WriteLn() ;
      WriteString('El numero ') ;
      WriteCard(VALOR, 4) ;
      WriteString(' no esta en la lista ' ) ;
        WriteCard(ap, 3) ;
      END
      WriteLn() ;
      WriteString(' Deseas continuar? (s/n): ') ;
      Read(mas) ;
    END ;
  END Busca.
```

```
program busca ;
(* Codificacion del programa de busqueda lineal en el lenguaje Pascal *)
(* Los "comentarios" en Pascal se encierran entre el par
   parentesis-asterisco  y asterisco-parentesis  *)
const INICIO = 1 ; FIN = 10 ;
var LISTA : array [INICIO..FIN] of integer ; ap, i, band, VALOR : integer ;
    mas : char ;
begin
(* Por desgracia en Pascal no existe una manera de darle valores
   iniciales a un arreglo. *)
LISTA[1] := 11  ;  LISTA[2] :=  9  ;  LISTA[3] := 27  ;  LISTA[4] := 32 ;
LISTA[5] := 52  ;  LISTA[6] := 46  ;  LISTA[7] :=  0  ;  LISTA[8] := 94 ;
LISTA[9] := 88  ;  LISTA[10] := 26 ;
mas := 's' ;
while ( mas =  s' ) do
begin
    band := 0 ;    (* baja la bandera *)
    ap := INICIO ;
    writeln ; writeln ;
    write( 'La lista es: ') ;
    for i := INICIO to FIN do write( LISTA[i], ' ' ) ;
    writeln ; writeln ;
    write( 'DAME EL VALOR A BUSCAR: ') ;
    readln( VALOR ) ;
    write( VALOR ) ;
(* Iteracion condicional para la busqueda *)
    while (ap <= FIN) AND (band = 0) do
        if LISTA[ap] = VALOR then band := 1   (* lo encontre *)
                             else ap := ap + 1 ;
    if band = 1 then
    begin  writeln ; writeln( 'ENCONTRE EL NUMERO ', VALOR,
                    ' EN LA LOCALIDAD No. ', ap )
    end else
    begin writeln ; writeln( 'EL NUMERO ', VALOR,
                         ' NO ESTA EN LA LISTA')
    end ;
    writeln ;
    write( 'DESEAS CONTINUAR? (s/n): ') ;
    readln ( mas ) ;
    write( mas ) ;
end ;
writeln ; writeln ;
writeln( ' ADIOS. ' ) ;
end.
```

```
% Programa de busqueda lineal, escrito en PROLOG
ListaInicial(11, 9, 27, 32, 52, 46, 0, 94, 88, 26).
LimitesIniciales(1, 10).

ListaInicial(Lista),
LimitesIniciales(Inicio, Fin),
Busqueda(Lista, Inicio, Fin)?
% Iteracion principal
Busqueda(Lista, Inicio, Fin):-
    Ciclo(Lista, Inicio, Fin, 's'),
    write("¿Deseas continuar? (s/n):"),
    read(Mas),
    Ciclo(Lista, Inicio, Fin, Mas).

% Iteracion mientras se desee seguir buscando
Ciclo(Lista, Inicio, Fin, 's'):-
    writeln("Dame el valor a buscar "),
    read(Valor),
    Busca(Lista, Inicio, Fin, Valor).

Ciclo(Lista, Inicio, Fin, Valor, _).

% Funcion de busqueda
Busca([Valor|Resto], Inicio, Fin, Valor):-
    write(" Encontre el numero "),
    write(Valor),
    write(" en la localidad "),
    write(Inicio).

% Aqui se busca a lo largo de la lista
Busca([X|,Resto], Inicio, Fin, Valor):-
    X <> Valor,
    Inicio <= Fin,
    Busca(Resto, Inicio+1, Fin, Valor).

Busca(Lista, Inicio, Fin, Valor):-
    Inicio > Fin,
    write(" El numero "),
    write(Valor),
    write(" no esta en la lista").
```

```
! Codificacion de la "version condensada" de la busqueda lineal
! en el lenguaje FOREST, que no usa la "bandera".
integer LISTA(10)
integer INICIO, FIN, VALOR, ap, xi
!
! Valores iniciales
data LISTA / 11, 9, 27, 32, 52, 46, 0, 94, 88, 26 /
data INICIO, FIN,  xi
   /    1  , 10 , 1Hs  /
!
mientras( xi = "s" )  ! iteracion principal
comienza
  ap = INICIO
  write(6,100) ( LISTA(i), i = INICIO, FIN )
  100 format( //, " La lista es: ", 50i3 )
  write(6,110) ; 110 format(//," DAME EL VALOR A BUSCAR: ")
  read(5,120) VALOR  ;  120 format(i3)
  write(6,120) VALOR
  ! iteracion condicional para la busqueda: un solo renglon
  mientras( LISTA(ap) <> VALOR  &  ap <= FIN )  ++ap
!
  si( LISTA(ap) = VALOR ) write(6,130) VALOR, ap
  otro write(6,140) VALOR
!
  130 format(//," ENCONTRE EL NUMERO", i3, " EN LA LOCALIDAD No.", i3 )
  140 format(//," EL NUMERO", i3, " NO ESTA EN LA LISTA" )
  write(6,150)   ;  150 format(//, " DESEAS CONTINUAR? (s/n):" )
  read(5,160) xi ;  160 format( 1a1 )
  ap = INICIO
termina  ! fin de la iteracion principal
write(6,170)  ;  170 format(//, " ADIOS." )
end
```

```
rem codificacion en CBASIC del programa para multiplicar matrices
100
input " Dame el numero de renglones de la matriz A: "; m%
input " Dame el numero de columnas de la matriz  A: "; n%
input " Dame el numero de columnas de la matriz  B: "; p%
if m% <= 0 or n% <= 0 or p% <= 0 then print "NO PUEDE SER..." : goto 100
dim A(m%,n%), B(n%,p%), C(m%,p%)
print : print
rem se leen los valores de las matrices
for i% = 1 to m%
  for j% = 1 to n%
    print "valor de A("; i%; "," ; j%; "): ";
    input A(i%,j%)
    print
  next j%
next i%
print : print : print
for i% = 1 to n%
  for j% = 1 to p%
    print "valor de B("; i%; "," ; j%; "): ";
    input B(i%,j%)
    print
  next j%
next i%
print : print : print
print "La matriz producto C("; m%; ","; p%; ") es: "
print : print : print
rem comienza el calculo.
for i%= 1 to m%
    for j% = 1 to p%
      C(i%,j%) = 0
      for k% = 1 to n%
        C(i%,j%) = C(i%,j%) + A(i%,k%) * B(k%,j%)
      next k%
    next j%
next i%
rem se imprimen los resultados.
for i% = 1 to m%
  for j% = 1 to p%
    print C(i%,j%) ;
  next j%
  print
next i%
stop
end
```

```
include <stdio.h>
   /*   Problema de las ocho damas   */
   /*   codificado en el lenguaje C  */
  int baja[8], sol, sube[8], vector[8] ;  /* estructuras de datos para */
                                          /* almacenar la informacion  */
                                        /* de las diagonales "amenazadas" */
  main()
  {
  int col, ren, result, ult ;
  int numero = 6 ;   /* numero de soluciones que se desea imprimir */

  ult = result = 0 ;
  ren = col = sol = 0 ;
  coloca(&ren,&col) ;  /* comienza la busqueda de soluciones */
  do {
     while( col < 7 )  /* trata de colocar la proxima dama */
      {
       if( result == 0 ) {
         ++col ;  /* ya se encontro una solucion, prepararse  */
                  /* para la siguiente.  */
         ren = -1 ;
       }
  /* verificar si hay lugar libre para la nueva dama */
       libre( &ren, &col, &result ) ;
       if( result == 1 ) atras( &ren, &col, &ult ) ; /* regresar a la     */
                                                     /* posicion anterior */
       else   coloca( &ren, &col ) ;
     }
  /* se encontro una solucion: imprimirla y buscar la siguiente  */
     imprime() ;
     atras( &ren, &col, &ult ) ;
     result = 1 ;
   }
   while( ult == 0  &&  sol < numero ) ;
  }

 coloca( ren,col )    /* coloca una dama en el tablero  */
  int *ren, *col ;
  {
  vector[*col] = *ren ;
  sube[*col]   = *ren - *col ;
  baja[*col]   = *ren + *col ;
  }
```

```
libre( ren, col, result )   /* averigua si hay una posicion libre */
                            /* en la columna actual de la dama    */
int *ren, *col, *result ;
{
int j ;
while( *ren < 7 )
    {
    *result = 0 ;
    ++*ren ;
    for( j = 0 ; j < *col ; ++j )
       if(     ( sube[j] == ( *ren - *col ) ) ||
               ( baja[j] == ( *ren + *col ) ) ||
               vector[j] ==   *ren          )   {

                                        *result = 1 ; /* abandona el  */
                                        break ;       /* 'for'        */
       }
    if( *result == 0 ) break ; /* abandona la iteracion 'while' */
 }
}
```

```
atras( ren, col, ult )   /* rutina para hacer 'backtrack' */
int *ren, *col, *ult ;
{
--*col ;
if( *col >= 0 )
  {
  *ren = vector[*col] ;
  vector[*col] = 0 ;
}
else   *ult = 1 ;  /* avisa que se encontro la ultima solucion */
}

imprime()   /* rutina que imprime una solucion */
 {
 int tabl[8][8], i, j ;
 for( i = 0 ; i < 8 ; ++i )
    for( j = 0 ; j < 8 ; ++j )
       tabl[i][j] = 0 ;

 ++sol ;
 if( sol % 6  == 1 ) printf( "\n\n\n\n\n\n" ) ;  /* imprime seis tableros */
 for( i = 0 ; i < 8 ;  ++i )                     /* por hoja              */
    tabl[ vector[i] ][i] =  i + 1 ;

 printf( "\n      sol. num. %d\n\n" , sol ) ;
 for( i = 0 ; i < 8 ; ++i ) {
    for( j = 0 ; j < 8 ; ++j )
       printf( " %d ", tabl[i][j] ) ;
    printf( "\n" ) ;
 }
 printf( "\n\n\n\n\n" ) ;    /* cambios de renglon */

 for( i = 0 ; i < 8 ; ++i )
    tabl[ vector[i] ] [i] = 0 ;
}
```

Ejemplo de los resultados del programa de las ocho damas.
Se muestra un tablero con ceros en las casillas vacías, y un número que indica dónde se colocó una dama.

```
      sol. núm. 1

1  0  0  0  0  0  0  0
0  0  0  0  0  0  7  0
0  0  0  0  5  0  0  0
0  0  0  0  0  0  0  8
0  2  0  0  0  0  0  0
0  0  0  4  0  0  0  0
0  0  0  0  0  6  0  0
0  0  3  0  0  0  0  0

      sol. núm. 2

1  0  0  0  0  0  0  0
0  0  0  0  0  0  7  0
0  0  0  4  0  0  0  0
0  0  0  0  0  6  0  0
0  0  0  0  0  0  0  8
0  2  0  0  0  0  0  0
0  0  0  0  5  0  0  0
0  0  3  0  0  0  0  0

      sol. núm. 3

1  0  0  0  0  0  0  0
0  0  0  0  0  6  0  0
0  0  0  0  0  0  0  8
0  0  3  0  0  0  0  0
0  0  0  0  0  0  7  0
0  0  0  4  0  0  0  0
0  2  0  0  0  0  0  0
0  0  0  0  5  0  0  0

      sol. núm. 4

1  0  0  0  0  0  0  0
0  0  0  0  5  0  0  0
0  0  0  0  0  0  0  8
0  0  0  0  0  6  0  0
0  0  3  0  0  0  0  0
0  0  0  0  0  0  7  0
0  2  0  0  0  0  0  0
0  0  0  4  0  0  0  0
```

```
sol. núm. 5

0 0 0 0 0 6 0 0
1 0 0 0 0 0 0 0
0 0 0 0 5 0 0 0
0 2 0 0 0 0 0 0
0 0 0 0 0 0 0 8
0 0 3 0 0 0 0 0
0 0 0 0 0 0 7 0
0 0 0 4 0 0 0 0

sol. núm. 6

0 0 0 4 0 0 0 0
1 0 0 0 0 0 0 0
0 0 0 0 5 0 0 0
0 0 0 0 0 0 0 8
0 2 0 0 0 0 0 0
0 0 0 0 0 0 7 0
0 0 3 0 0 0 0 0
0 0 0 0 0 6 0 0
```

Palabras y conceptos clave

En esta sección se agrupan las palabras y conceptos de importancia que se estudiaron en el capítulo, con el objetivo de que el lector pueda hacer una autoevaluación que consiste en ser capaz de describir con cierta precisión lo que cada término significa, y no sentirse satisfecho hasta haberlo logrado. Se muestran en el orden en que se describieron o se mencionaron.

ALGORITMO DE BÚSQUEDA
PRUEBA DE UN PROGRAMA
TABLA DE VERDAD
REFINAMIENTOS PROGRESIVOS
CARGA SEMÁNTICA
VECTOR
ÍNDICE
ASIGNACIÓN
MÓDULO
MODULARIDAD
PASO DE PARÁMETROS
DISEÑO ESTRUCTURADO
STUB
USUARIO DE UN SISTEMA
DOCUMENTACIÓN
MANUALES
COMENTARIOS EN UN PROGRAMA
PRUEBA FORMAL
CÁLCULO PROPOSICIONAL
VALOR DE VERDAD
TAUTOLOGÍA

Ejercicios

1. Modifique el pseudocódigo final de la búsqueda lineal para que encuentre la última aparición del número pedido, y no la primera.
2. Escriba programas en pseudocódigo para resolver los problemas 4, 5 y 6 del capítulo 2, y observe cómo estas soluciones son casi triviales cuando se comparan con el esfuerzo requerido para lograrlas en lenguaje de máquina.

3. Escriba el pseudocódigo refinado de un programa que tome un conjunto no muy grande de números y calcule su promedio. Suponga que los números están dentro de un vector y aproveche el recorrido sobre él para ir tomando los datos para el cálculo, así como para determinar su longitud.
4. Se llama "cuadrado mágico" a un arreglo de números en forma de matriz en el que la suma de cada uno de sus renglones y columnas, así como de las dos diagonales principales, da el mismo número.

   Por ejemplo,

   2 9 4
   7 5 3
   6 1 8

   es un cuadrado mágico, y cualquier matriz cuadrada con elementos iguales también lo es.

   Haga y refine un programa en pseudocódigo que determine si una matriz dada es o no un cuadrado mágico. (No confundir esto con un programa que genere uno de estos cuadrados, lo cual es mucho más difícil.)
5. Haga un programa en pseudocódigo para simular una máquina de Turing. (O sea, describa lo que se tiene que hacer, con detalle, para que una máquina de Turing siga las instrucciones codificadas en su tabla.)
6. Diseñe el pseudocódigo de un pequeño sistema de reservaciones aéreas. Defina módulos que se encarguen de las funciones básicas (buscar un lugar libre dentro de un vuelo, reservar un lugar, buscar un nombre en la lista de pasajeros, etc.) y organícelos dentro de un programa principal que controle la operación.
7. Tome el pseudocódigo (aunque allí no se llamaba así) del ensamblador expuesto en el capítulo 4 y refínelo lo más posible.
8. Escriba el pseudocódigo de un macroprocesador como el delineado en el capítulo 4.
9. Refine el pseudocódigo de un cargador como el mostrado en el capítulo 4.
10. Como un tablero de damas es cuadrado, las soluciones del problema de las ocho damas muestran simetría con respecto a la rotación de 90 grados; esto significa que las soluciones se pueden organizar en grupos de cuatro, que representarán las cuatro aparentemente diferentes posiciones que verían observadores en cada lado del tablero. Modifique el pseudocódigo del texto para encontrar estos grupos de cuatro soluciones, e identifique de esta forma las soluciones únicas, que son independientes de la rotación.

Referencias para el capítulo 7

[BROF75] Brooks, Frederick, Jr., *The Mythical Man-Month*, Addison Wesley, Massachusetts, 1975.
En este libro se discuten temas de sumo interés para todo aquel involucrado de alguna manera en proyectos de programación. Trata desde aspectos históricos del desarrollo de grandes sistemas (el autor, Brooks, fue el responsable del desarrollo del enorme sistema operativo IBM OS/360), hasta enfoques interesantes acerca de la productividad en los programadores, la programación estructurada, etcétera.

[DEMT79] De Marco, Tom, *Concise Notes on Software Engineering*, Yourdon Press, Nueva York, 1979.
Este pequeño libro sirve como introducción a los cursos y seminarios que la academia Yourdon imparte en nivel internacional. En sus 80 páginas discute el plan de diseño estructurado que propone e impulsa esa combinación de compañía-academia-editorial.

[FRIF84] Friedman, Frank y Elliot Koffman, *FORTRAN. Introducción al lenguaje y resolución de problemas con programación estructurada*, Fondo Educativo Interamericano, México, 1984.
Traducción de un buen libro sobre este lenguaje, el más antiguo, que sigue siendo de amplio uso. Este texto lo emplea como vehículo para enseñar programación estructurada, y explica el uso de FORTRAN 77. Todos los ejemplos están codificados en español. Este libro tiene una estructura (y subtítulo) similar a uno de Pascal, escrito por uno de los coautores, [KOFE86], mencionado en el capítulo 8.

[FRIF86] Friedman, Frank y Elliot Koffman, *BASIC*, Addison-Wesley Iberoamericana, México, 1986.
Traducción de un libro sobre este lenguaje de programación, propio supuestamente para principiantes. Debido a que no existe un único estándar de este lenguaje, el lector debe esperar ciertas diferencias entre el BASIC aquí expuesto y el que incluya la computadora en particular con la que trabaje.

[GRID81] Gries, David, *The Science of Programming*, Springer-Verlag, Nueva York, 1981.
Libro introductorio sobre diseño de programas que considera esta tarea como una ciencia formal y, por ende, sujeta a razonamientos matemáticos en cada una de sus fases. Dedica los primeros seis capítulos a un estudio de la lógica mate-

mática requerida para sustentar demostraciones y luego define poco a poco todo lo necesario para poder construir programas completos. Tal vez la siguiente frase, tomada de uno de los principios metodológicos que propone, resuma todo el enfoque del texto: "Un programa y su prueba deben desarrollarse en conjunto, y la prueba debe llevar la delantera".

[GROP86] Grogono, Peter, *Programación en Pascal*, Addison-Wesley Iberoamericana, México, 1986.
Traducción de un libro muy exitoso sobre este lenguaje, que se ha convertido en virtual estándar para cursos de programación estructurada.

[HARD87] Harel, David, *Algorithmics: the Spirit of Computing*, Addison-Wesley, Massachusetts, 1987.
Interesante libro introductorio sobre ciencias de la computación que trata sobre la génesis de los métodos algorítmicos así como de temas relacionados como corrección, eficiencia, teoría de la computabilidad, concurrencia y paralelismo. Tiene un capítulo completo sobre notas bibliográficas.

[HOAC69] Hoare, Charles, "An axiomatic Basis for Computer Programming", en *Communications of the Association for Computing Machinery*, vol. 12, núm. 10, octubre, 1969.
Este artículo comienza señalando que "la programación de computadoras es una ciencia exacta, en el sentido que todas las propiedades de un programa y las consecuencias de su ejecución en un entorno dado pueden, en principio, ser extraídas del programa mismo por medio de razonamientos puramente deductivos"; es una introducción bastante accesible —aunque requiere conocimientos de matemáticas— al tema de la prueba de corrección de programas. El autor, muy conocido en el mundo de la computación, es inventor de uno de los mejores algoritmos de ordenamiento (*Quick-sort*).

[JACM75] Jackson, M. A., *Principles of Program Design*, Academic Press, Londres, 1975.
Lo que podríamos llamar "escuela europea de diseño" está representada en un primer nivel por la metodología que propone Jackson, y que ha tenido un gran auge, sobre todo en la programación de tipo administrativo. El libro propone diagramas especiales para describir la estructura de un sistema, y codifica la mayoría de los ejemplos en el lenguaje COBOL.

El autor produjo luego otro texto, esta vez no dedicado al diseño de programas, sino de sistemas completos: *Systems Development*, Prentice-Hall, 1983.

[KERB78] Kernighan, Brian y P. Plauger, *The Elements of Programming Style*, McGraw-Hill, Nueva York, 1978.
Excelente libro sobre el estilo de programar, que considera la actividad de escribir programas como cercana a la de escribir cuentos o novelas cortas y que, por ende, propicia el surgimiento de un estilo propio de escritura. Está lleno de ejemplos de cómo programar y cómo no programar, tomados de múltiples libros de texto sobre computación. Es necesario saber un poco de FORTRAN o PL/I para entenderlo totalmente. Existe traducción al español.

[KERB85] Kernighan, Brian y Dennis Ritchie, *El lenguaje de programación C*, Prentice-Hall Hispanoamericana, México, 1985.
Traducción del libro original donde se describe el lenguaje C, en el entorno del sistema operativo en el que funciona (Unix). El libro es interesante porque está escrito por los autores del lenguaje, que también son coautores de Unix. Se acaba de publicar una segunda edición, ampliada (1988).

[KNUD73] Knuth, Donald, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Massachusetts, 1973.
Este tercer volumen (y hasta la fecha el último) de la gran enciclopedia que Knuth comenzara a escribir a finales de la década de 1960 trata sobre los algoritmos de ordenamiento y búsqueda. Los otros dos volúmenes tratan sobre algoritmos fundamentales (el primero) y algoritmos seminuméricos (el segundo), y aún se consideran verdaderos clásicos en la literatura computacional, tanto por la calidad y profundidad con la que se estudian los temas como por el enfoque matemático que se les da.

Knuth dijo en una entrevista reciente que se propone continuar con su obra fundamental, una vez que terminó ya seis volúmenes (no previstos por él) sobre tipografía matemática. Existe traducción al español del primer volumen. Knuth es coautor de un nuevo libro sobre matemáticas para la computación: *Concrete Mathematics*, Addison-Wesley, 1989.

[LEVG80] Levine, Guillermo, *Manual para el usuario del sistema FOREST (FORtran ESTructurado)*, Departamento de ingeniería, Universidad Autónoma Metropolitana-Iztapalapa, México, 1980.
El autor se propuso escribir un preprocesador para convertir FORTRAN en un mejor lenguaje de programación, más

adecuado para la programación y el diseño estructurados. Durante varios años fue empleado por la comunidad estudiantil de la universidad, hasta que Pascal y otros lenguajes estructurados se hicieron de uso común.

[LOGI70] *Lógica*, National Council of Teachers of Mathematics, Col. Temas de matemáticas, vol. 12, Trillas, México, 1970.
Traducción de un libro elemental sobre lógica matemática, orientado hacia profesores de matemáticas en escuelas primarias. Resulta una introducción muy accesible a este tema, pues tiene tan sólo sesenta páginas.

[ORRK77] Orr, Kenneth, *Structured Systems Development*, Yourdon Press, Nueva York, 1977.
Libro en el que se expone el método de diseño estructurado de sistemas que se conoce como de "Warnier-Orr", de gran aceptación en algunos círculos. Warnier es un profesor francés, inventor del sistema de diagramación que se emplea, y que constituye el rasgo distintivo de la metodología propuesta. Como el libro no es muy fácil de conseguir, se recomiendan también los artículos "Structured Programming with Warnier-Orr Diagrams", de David Higgins, en *Byte*, diciembre de 1977 (Parte I), enero de 1978 (Parte II).

[PHIA87] Philippakis, Andreas y Leonard Kazmier, *COBOL estructurado*, McGraw-Hill, Colombia, 1987
Traducción de la tercera edición de un exitoso y amplio libro que combina el lenguaje COBOL con las técnicas de programación estructurada. Tiene una gran cantidad de ejemplos y diagramas de flujo estructurados, y esta nueva edición resalta en color las construcciones que han cambiado o mejorado en este lenguaje, todavía de muy amplio uso.

[SUPP66] Suppes, Patrick, *Introducción a la lógica simbólica*, CECSA, México, 1966.
Traducción de un excelente libro sobre lógica matemática, teoría de conjuntos y teoría de la demostración y la inferencia. Todo esto tratado en un nivel muy accesible. En el último capítulo se estudia en forma introductoria el método axiomático en matemáticas.

[TARA68] Tarski, Alfred, *Introducción a la lógica y a la metodología de las ciencias deductivas*, Nueva Ciencia - Nueva Técnica, Espasa Calpe, Madrid, 1968.
Traducción de la obra de uno de los grandes lógicos contemporáneos. Explica con detalle el funcionamiento del cálculo proposicional y las teorías de clases y relaciones, para luego

pasar al estudio de temas más avanzados de metodología formal y matemática.

[WEIG71] Weinberg, Gerald, *The Psychology of Computer Programming*, Van Nostrand Reinhold, Nueva York, 1971.
Fascinante estudio sobre la "programación de computadoras como una actividad humana", que discute tanto aspectos relativamente técnicos como motivaciones y consideraciones personales de los involucrados en las diversas facetas del trabajo con computadoras (basadas en estudios y entrevistas hechas por el autor y otros). Incluye, al final de cada capítulo, preguntas y material de discusión dirigidas a programadores, jefes de proyecto y gerentes de organización. Entre las innovaciones que se proponen resalta la muy benéfica idea de la "programación sin ego", en donde se plantea que los programas que el programador escribe no son necesariamente reflejo directo de su personalidad y que, por tanto, debe ser posible criticarlos técnicamente sin por ello sentir que se hace un ataque personal.

[WIRN71] Wirth, Niklaus, "Program Development by Stepwise Refinements", en *Communications of the Association for Computing Machinery*, vol. 14, núm. 4, abril, 1971.
Es en este artículo donde Wirth propone que "la actividad creativa de la programación debe diferenciarse de la actividad de la codificación", y muestra a grandes rasgos su método de refinamientos progresivos, precisamente poniendo como ejemplo una solución al problema de las ocho damas.

[WIRN76] Wirth, Niklaus, *Algorithms + Data Structures = Programs*, Prentice-Hall, New Jersey, 1976.
En este libro, Wirth desarrolla toda una metodología de programación estructurada, enriquecida (y a veces también oscurecida) con aplicaciones del lenguaje de programación Pascal que él mismo diseñara. Enfatiza la interrelación que debe existir siempre en un "buen" programa entre los datos y los algoritmos.
Existe traducción al español.

Wirth desarrolló posteriormente el lenguaje de programación Modula-2, y escribió una segunda edición del libro, esta vez dando todos los ejemplos en el nuevo lenguaje.

8

La codificación en la programación estructurada: FORTRAN y Pascal

8.1 Introducción

Si el lector ha estudiado con cuidado los capítulos anteriores, sin mayor problema podrá dedicar sus esfuerzos a la parte final de la metodología que se ha explicado, y que consiste en expresar los algoritmos ya diseñados, en algún lenguaje de programación en particular.

El plan de este capítulo es, pues, como sigue: se tomarán los ejemplos y las ideas anteriormente esbozadas (y otras nuevas) y se desarrollarán hasta el punto de la codificación final en varios lenguajes de programación. Se decidió escoger Pascal y FORTRAN para esto debido a la creciente popularidad del primero y a la accesibilidad del segundo.

En términos generales, FORTRAN no facilita las tareas de la programación estructurada, pues carece de los recursos sintácticos para ello. Se escogió precisamente para ilustrar al lector la manera de subsanar, usando la metodología del libro, sus principales puntos débiles en este campo.

Pascal, por su parte, es un lenguaje de programación que sí permite y alienta la programación estructurada, por lo que será más sencillo codificar los algoritmos con él.

Antes de comenzar, advertimos seriamente al lector que no nos proponemos enseñarle a programar en Pascal ni en FORTRAN, sino que se emplea-

rán estos lenguajes como ejemplos de codificación. En la bibliografía del capítulo anterior se propusieron algunos libros y manuales (entre muchos otros) sobre lenguajes de programación, por lo que nos limitaremos a explicar las características de los dos lenguajes de tal forma que permitan la comprensión de lo que aquí se dice, lo cual es muy diferente de aprenderlos a fondo. Además de las referencias citadas en el capítulo anterior, [GROP86] de Pascal, y [FRIF84] de FORTRAN, se proponen (entre tantos posibles) los libros [JONW82], [KERB81] y [KOFE86] para el lenguaje Pascal, así como [DAVG86] y [SCHW74] para FORTRAN. [JENK74] es la referencia estándar original para el lenguaje Pascal.

Se sugiere usar este capítulo para practicar los conocimientos que el lector haya adquirido previamente en FORTRAN o Pascal, de forma tal que le sirva para integrarlos a la metodología hasta aquí explicada.

8.2 Estructuras fundamentales de control

Familia de lenguajes de programación

Es posible clasificar los lenguajes de programación como los mencionados en el anexo 6.4 (burdamente, es cierto) en dos familias, dependiendo de la facilidad que otorguen al programador para expresar algoritmos de forma estructurada. La guía serán, simplemente, las estructuras mínimas de control que se aprendieron en el capítulo 6.

Desde este punto de vista, BASIC, COBOL y FORTRAN forman una familia, mientras que Ada, ALGOL, C, Modula-2, Pascal y PL/I forman otra, más "estructurada". APL, LISP y Prolog son ejemplos de lenguajes cuya filosofía no está basada en procedimientos, sino en conceptos matemáticos formales diferentes de los explicados en este libro, por lo que quedan fuera de este enfoque.

Nuestra primera tarea es, pues, aprender a expresar las estructuras de secuenciación, selección e iteración condicional en lenguajes de las dos familias. Todos ellos permiten la secuenciación de una forma similar, esto es, un enunciado en cada renglón. Sin embargo, hay diferencias; FORTRAN y COBOL requieren formato fijo, lo que significa que los enunciados deben ser escritos a partir de ciertas posiciones del renglón, ya que de otra manera el compilador no los reconoce correctamente. Es más, en FORTRAN se permite sólo un enunciado por renglón, (que deberá forzosamente comenzar en la columna 7) mientras que en todos los demás lenguajes existen maneras de escribir varios, separándolos mediante algún signo especial de puntuación. Es decir, si una expresión de FORTRAN demasiado larga ocupa varios renglones, es obligatorio escribir un signo de continuación en la columna 6 en todos ellos a partir del segundo, para avisar al compilador que se trata todavía de la misma instrucción.

En el caso de casi todos los demás lenguajes, una instrucción puede extenderse a varios renglones sin necesidad de un signo de continuación. Esto se

debe a que toda expresión tiene un terminador, que indica al compilador dónde termina una y dónde comienza la siguiente.

Secuenciación

Entonces, la expresión en pseudocódigo

```
e1; e2; e3
```

donde los enunciados son, por ejemplo, instrucciones de asignación, se expresará en FORTRAN como:

```
ALFA = 1
BETA = 2
ZETA = 3
```

y en Pascal como

```
alfa := 1  ;  beta := 3  ;  zeta := 3  ;
```

Es muy importante tomar en cuenta que Pascal diferencia claramente los diversos usos del signo de igualdad, y cuando se trata de la asignación, como en el ejemplo, exige que se use el doble símbolo := (véase la pág. 224).

Además, Pascal requiere que todas y cada una de las variables que se usen en un programa estén declaradas al inicio, asunto que se tratará cuando se hable de las estructuras de datos.

Cuando se desea expresar la secuenciación de varios enunciados de tal forma que aparezca como un solo enunciado elemental se emplean, en pseudocódigo, los metaparéntesis comienza y termina El primer problema en FORTRAN es que se carece de esta facilidad sintáctica, por lo que se debe simular por otros medios, lo cual se hará un poco más adelante. Pascal, por lo pronto, dispone de las palabras **begin** y **end**, que se usan de la misma manera que los paréntesis del pseudocódigo*.

Hay un detalle que requiere explicación. En Pascal, el punto y coma se usa como separador, no como terminador, lo que significa que sirve para indicar al compilador la diferencia entre uno o más enunciados. Si el último enunciado antecede directamente a una palabra clave de Pascal, ya no será necesario separarlo de lo que sigue con un punto y coma, puesto que el compilador notará que la siguiente palabra es especial, y determinará que el enunciado terminó.

* Por razones de claridad seguiremos subrayando las palabras clave en los programas en pseudocódigo, aunque ya no se hará en los escritos en FORTRAN o Pascal, para que el lector tenga claro que no es necesario subrayarlas en los programas ya codificados.

Es decir, la expresión en pseudocódigo

```
.
.
.
comienza
   e1
   e2
   e3
termina
.
.
.
```

se expresará en Pascal de cualquiera de estas dos formas:

```
.
.
.
begin
alfa := 1 ;
beta := 2 ;
zeta := 3
end
.
.
.
```

```
.
.
.
begin
alfa := 1 ; beta := 2 ; zeta := 3
end
.
.
.
```

Selección

La estructura **si** ... **entonces** ... **otro** del pseudocódigo tiene expresión similar en Pascal (`if... then ... else`), aunque no en FORTRAN (que dispone del `if`, pero no del `else`), donde se vuelve necesario simularla, o usar trucos de programación como el siguiente.

Estas dos expresiones son equivalentes:

FORTRAN	Pascal
`IF ( ALFA .EQ. 85 ) BETA = 38`	`if alfa = 85 then beta := 38`
`IF ( ALFA .NE. 85 ) BETA = 10`	`else beta := 10 ;`

El truco consistió en aprovechar nuestro conocimiento de las condiciones booleanas para simular lo dicho por el pseudocódigo, ya que no puede ser que ALFA sea igual (.EQ.) a cierto valor, siendo al mismo tiempo diferente (.NE.) de él. Por tanto, se repite la pregunta, invirtiendo el signo.

¿Qué hubiera sucedido si el programa en FORTRAN dijera

```
IF ( ALFA .EQ. 85 ) BETA = 38
BETA = 10
```

en lugar de lo anterior? Pues que si la variable ALFA vale 85, entonces primero se da el valor 38 a BETA para, inmediatamente después, cambiarlo por 10, lo cual es un error. Todo esto es porque FORTRAN carece de la cláusula **else**.

En el caso de Pascal la expresión ocupa dos renglones, y está separada de lo que venga después mediante el punto y coma. Es vital entender que el separador no debe ir después de la proposición **beta := 38**, porque entonces la cláusula **else** estaría fuera de contexto (o sea, no estaría dentro del alcance sintáctico del **if**).

Al surgir la necesidad de formular expresiones más complejas es cuando se observa con claridad la ventaja del pseudocódigo y la consiguiente debilidad de los lenguajes de programación no estructurados, como en el siguiente ejemplo:

```
si  alfa  =  88 entonces comienza
                          beta = 6
                          zeta = 7
                         termina
                otro     comienza
                          beta = 0
                          zeta = 2
                         termina
```

que en Pascal se escribe virtualmente igual:

```
if alfa = 88 then begin
                    beta := 6 ;
                    zeta := 7
                  end
             else begin
                    beta := 0 ;
                    zeta := 2
                  end ;
```

mientras que para FORTRAN no basta siquiera con el truco anterior (¿por qué?), y es necesario simular toda la construcción, usando elementos no estructurados (véase la pág. 202) e introduciendo la construcción de etiqueta que se mencionó al hablar de los ensambladores:

```
      IF( ALFA .NE. 88 ) GOTO 100
C                   CLAUSULA 'THEN'
      BETA = 6
      ZETA = 7
      GOTO 200
C                   CLAUSULA ELSE
100   BETA = 0
      ZETA = 2
200   CONTINUE
```

Una etiqueta es simplemente un número que ocupa una posición entre las columnas 1 y 5, y que sirve como blanco de la instrucción GOTO, misma

que desvía el flujo de control de la secuencia que llevaba y lo dirige al renglón que comienza con la etiqueta en cuestión (que puede contener únicamente la palabra `CONTINUE`, que sirve como instrucción neutral, para simplemente ocupar el renglón y no dejarlo en blanco, lo cual sería ilegal).

Como resulta claro, el programa en FORTRAN perdió gran parte de la legibilidad del pseudocódigo, además de que tiene otro truco, bastante más sutil: hubo necesidad de invertir la condición (de ser `.EQ.` se convirtió en `.NE.`), para simular adecuadamente el flujo de control deseado. Recomendamos al lector no continuar hasta no haber entendido cabalmente esto.

De acuerdo con lo mencionado en la sección 7.6 respecto a los comentarios en un programa, se decidió incluir aquí dos, que forzosamente deben comenzar con la letra `C` en la columna 1 del renglón.

De la misma forma, los comentarios en Pascal, que pueden ocupar cualquier lugar que llene un espacio en blanco, se escriben rodeados del doble símbolo (* por la izquierda, y *) por la derecha, aunque empleen más de un renglón.

También se pudo haber codificado de esta forma:

```
      IF ( ALFA .EQ. 88 ) GOTO 100
C
      BETA = 0          CLAUSULA 'ELSE'
      ZETA = 2
      GOTO 200
C                       CLAUSULA 'THEN'
100   BETA = 6
      ZETA = 7
200   CONTINUE
```

sin necesidad de invertir la condición booleana; sólo que ahora el orden en que están escritas las proposiciones del entonces y el otro del pseudocódigo no se conserva, sino que se invierte.

En general, en FORTRAN siempre será conveniente invertir la condición booleana para conservar el orden estipulado en el pseudocódigo aunque, en realidad, esto dependerá del estilo de programación del lector.

La siguiente es una tabla de operadores booleanos y sus inversos:

OPERADOR	EN FORTRAN	OPERADOR INVERSO EN FORTRAN
¿Es igual?	.EQ.	.NE.
¿Es diferente?	.NE.	.EQ.
¿Es mayor?	.GT.	.LE.
¿Es menor?	.LT.	.GE.
¿Es mayor o igual?	.GE.	.LT.
¿Es menor o igual?	.LE.	.GT.

Obsérvese que el inverso de "¿es mayor?" no es "¿es menor?", como podría parecer en principio, sino "¿es menor o igual?" (¿Cuál es la razón?).

Cuando se trata de negar una condición booleana compuesta (del tipo de las usadas en el ejemplo de la búsqueda lineal, (véase la pág. 218) se vuelve necesario usar las llamadas *leyes de De Morgan*, que pueden ser expuestas así:

- La negación de la conjunción de A y B es la disyunción de las negaciones de A y B.

Y su ley dual:

- La negación de la disyunción de A y B es la conjunción de las negaciones de A y B.

Como un ejemplo se puede negar el enunciado "Son las seis o hace frío" de la siguiente manera: "Ni son las seis ni hace frío"; o sea, "No son las seis y no hace frío".

Ya que negar condiciones compuestas puede ser relativamente oscuro, sugerimos en su lugar el uso del operador .NOT., que simplemente antecederá a la condición completa, *encerrada ésta entre paréntesis* (para asegurar que el operador la afecta a toda ella y no sólo a su primer miembro).

De esta forma, la frase "Son las seis o hace frío" se negará así: "No es el caso que (son las seis o hace frío)", que es menos elegante, pero igual.

Para aplicar lo expuesto, considérese el siguiente pseudocódigo:

```
si hora = 6 y temperatura = frío
        entonces comienza
          alfa = 2; beta = 3
        termina
        otro
        comienza
          alfa = 10; beta = 20; zeta = 40
        termina
```

y su traducción a FORTRAN:

```
      IF( .NOT. ( HORA .EQ. 6 .AND. TEMP .EQ. FRIO ) ) GOTO 100
C                             CLAUSULA 'THEN'
      ALFA = 2
      BETA = 3
      GOTO 200
C                             CLAUSULA 'ELSE'
100   ALFA = 10
      BETA = 20
      ZETA = 40
200   CONTINUE
```

Iteración condicional

Pascal dispone de la instrucción `while`, que es parecida al `mientras` del pseudocódigo, por lo que no habrá mayor problema para usarla. La única diferencia es el uso obligado de la palabra `do` del lado derecho de la condición, de forma que ésta esté delimitada por un `while` a la izquierda y un `do` a la derecha, que actúan igual que los paréntesis usados en el pseudocódigo.

En FORTRAN nos veremos obligados, de nuevo, a simular la iteración condicional usando el `GOTO`.

Esta expresión en pseudocódigo

```
   .
   .
   .
mientras (alfa <> 10)
comienza
   beta = beta - 1
   alfa = alfa + 1
termina
   .
   .
   .
```

se escribe así en Pascal:

```
   .
   .
   .
while alfa <> 10 do
begin
    beta := beta - 1 ;
    alfa := alfa + 1
end ;
   .
   .
   .
```

y así en FORTRAN:

```
      .
      .
      .
100   CONTINUE
      IF( ALFA .EQ. 10 ) GOTO 200
```

```
C                  INICIO DE LA CLAUSULA ITERATIVA
      BETA = BETA - 1
      ALFA = ALFA + 1
      GOTO 100
C                  FIN DE LA CLAUSULA ITERATIVA
200   CONTINUE
        .
        .
        .
```

donde, una vez más, se invierte (niega) la condición original, y se construye la estructura por medios más primitivos. Consulte el lector la figura de la pág. 192 para aclarar aun más esto.

Se procede ahora a combinar y anidar estructuras (lo cual el lector ya debe dominar bien en pseudocódigo), y a codificarlas.

Tómese el ejemplo de la pág. 200, que decía:

```
si  C1  entonces comienza
                  si  C2  entonces e1
                          otro e6
                  mientras (C15) e9
                termina
        otro si C21 entonces comienza
                              e2
                              e33
                            termina
                  otro comienza
                         e11
                         e7
                       termina
```

y tradúzcase a Pascal, empleando condiciones y enunciados arbitrarios:

```
if alfa > 10 then begin
                    if beta < 0 then zeta := 1
                                else zeta := 7 ;
                    while gamma <= 25 do gamma:= gamma + 1
                  end
             else if delta = 8 then begin
                                      psi := 7 ;
                                      tau := 9
                                    end
                               else begin
                                      psi := -11 ;
                                      tau := -27
                                    end ;
```

Recomendamos al lector estudiar con cuidado la codificación en FORTRAN, porque es relativamente oscura:

```
      IF( ALFA .LE. 10 ) GOTO 260
C ------------ CLAUSULA 'THEN' DEL PRIMER 'IF'
C                  COMIENZA EL SEGUNDO 'IF'
      IF( BETA .LT.  0 ) ZETA = 1
      IF( BETA .GE.  0 ) ZETA = 7
C                  TERMINA EL SEGUNDO 'IF'
C                  COMIENZA EL 'WHILE'
240   IF( GAMMA .GT. 25 ) GOTO 250
      GAMMA = GAMMA + 1
      GOTO 240
C                  TERMINA EL 'WHILE'
250   CONTINUE
C                  TERMINA EL 'WHILE'
      GOTO 300
C ------------ CLAUSULA 'ELSE' DEL PRIMER 'IF'
260   CONTINUE
      IF( DELTA .NE. 8 ) GOTO 270
C                        CLAUSULA 'THEN' DEL TERCER 'IF'
      PSI = 7
      TAU = 9
      GOTO 300
C                        CLAUSULA 'ELSE' DEL TERCER 'IF'
270   CONTINUE
      PSI = -11
      TAU = -27
C ------------ TERMINA LA ESTRUCTURA...
300   CONTINUE
```

En este ejemplo se han empleado todos los trucos mencionados a lo largo del capítulo, pero aun así el programa parece muy pobre en comparación con el pseudocódigo o la codificación en Pascal. No es mucho lo que el programador puede hacer para lograr claridad, si debe trabajar con un lenguaje tipo FORTRAN o BASIC. Nuestra recomendación es ceñirse lo más posible a la guía del pseudocódigo, explicando mediante comentarios los pasos que se tuvieron que dar en la codificación.

Características propias de FORTRAN y Pascal

Llegó el momento de mencionar las particularidades indispensables de FORTRAN y Pascal que permitan escribir un programa completo y acabado. Se trata principalmente de las declaraciones de las estructuras de datos, sin entrar en demasiados detalles, sobre todo del lenguaje Pascal, que es especialmente rico en este campo.

Un programa en FORTRAN debe terminar con la palabra reservada `END`, que indica al compilador dónde acaba el programa fuente. No es forzoso que las variables que se empleen estén declaradas, porque se suponen de tipo entero todas las que comiencen con las letras de la I a la N, y de tipo real todas las demás. El programador puede, sin embargo, volver a declarar a discreción las variables que desee, por medio de las palabras `INTEGER` y `REAL`. Los vectores y arreglos deben ser declarados explícitamente por medio de la palabra `DIMENSION`, o bien incluidos en las declaraciones `INTEGER` o `REAL`.

Las instrucciones y formatos de lectura y escritura son particularmente elaboradas en FORTRAN, por lo que aquí se usarán las proposiciones `READ(*,*)` y `WRITE(*,*)` seguidas de las variables requeridas, dejando al lector la tarea específica de averiguar los requerimientos del compilador con el que vaya a trabajar. Muchas versiones de FORTRAN permiten estas instrucciones de entrada/salida, llamadas de *formato libre*.

Un programa en Pascal debe comenzar con la palabra clave `program`, seguida del nombre del programa y el punto y coma.

Las variables deben ser declaradas usando la palabra `var`, seguida de la lista de variables de un cierto tipo (separadas por comas), que terminará con dos puntos y la palabra `integer`, `real` o `char`.

Este último tipo (inexistente en FORTRAN) sirve para especificar cadenas de caracteres y es muy útil para manejar letras, como se verá más adelante.

Luego debe ir la palabra `begin`, que marca el inicio de las proposiciones ejecutables de Pascal, mismas que deben terminar con la palabra `end` seguida de un punto.

Como instrucciones de entrada/salida usaremos `readln`, `write` y `writeln`, dejando al lector la tarea de averiguar los detalles específicos que usa el compilador con el que va a trabajar.

Estos dos fragmentos de programa son equivalentes:

FORTRAN

```
DIMENSION ALFA (10)
INTEGER BETA, ZETA (25)
REAL LAMBDA, MU

   .
   .
   .

END
```

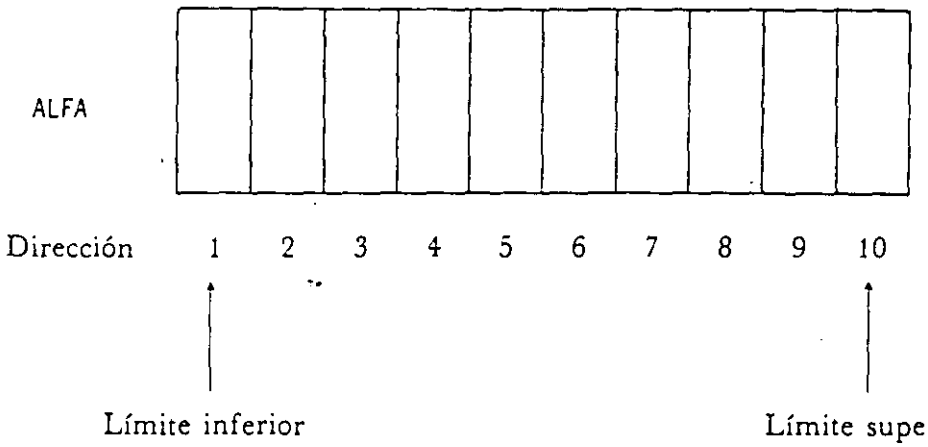
Pascal

```
Program ejemplo ;
var alfa: array [1..10] of integer ;
    zeta: array [1..25] of real ;
    beta: integer ;
    lambda, mu : real ;
begin

   .
   .
   .
end.
```

La codificación completa y final del cuarto refinamiento de la búsqueda lineal (pág. 223) aparece en el anexo 7.8 escrita en varios lenguajes, Pascal y FORTRAN entre ellos.

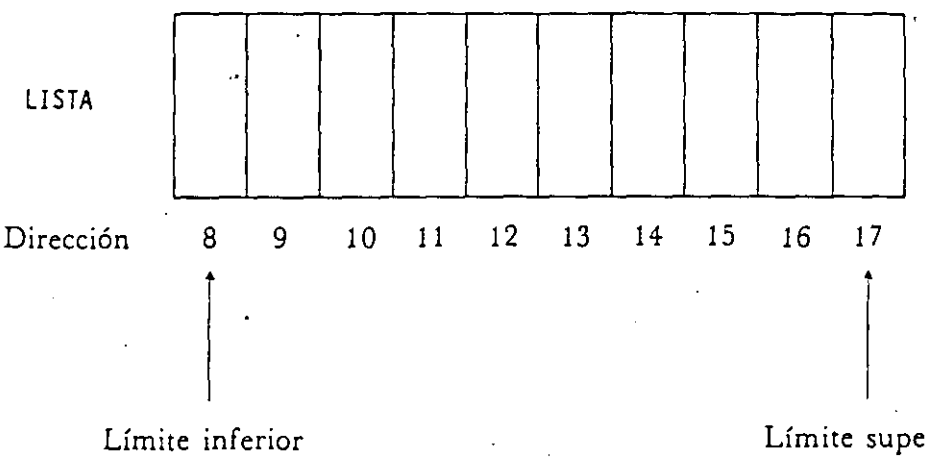
Obsérvese que Pascal permite la especificación de los límites inferior y superior de un vector, mientras que en FORTRAN todos los vectores y matrices tienen como límite inferior el número uno. Es decir, el vector ALFA de diez posiciones enteras está definido así:



mientras que esta declaración de Pascal no tiene equivalente directo en FORTRAN:

```
var LISTA: array [8..17] of integer;
```

porque representa el siguiente vector:



Si el compilador se encuentra en el programa una instrucción que diga, por ejemplo

```
LISTA[2] := 0;
```

marcará un mensaje de error (que tal vez aparezca durante la ejecución), ya que la casilla con dirección 2 del arreglo LISTA no está definida.

8.3 Estructuras adicionales de control

Se explicará ahora cómo usar algunas de las construcciones adicionales que ofrece Pascal, y que enriquecen considerablemente el poder expresivo de los programas. FORTRAN, como puede esperarse, se queda bastante atrás, pero puede perfectamente adaptarse (aunque con más esfuerzo) a casi cualquier tarea.

La construcción de pseudocódigo

```
repite
comienza
   e2
   e3
termina
hasta (condición)
```

tiene expresión inmediata en Pascal, de esta forma:

```
repeat
  alfa := alfa + 1 ;
  beta := beta - 1
until alfa = 10 ;
```

Nótese que las palabras **repeat** y **until** sirven como paréntesis, por lo que no es obligatorio delimitar los enunciados con **begin** y **end**.

En FORTRAN esto se dice así:

```
100   CONTINUE
      ALFA = ALFA + 1
      BETA = BETA - 1
      IF( ALFA .NE. 10 ) GOTO 100
```

Obsérvese que se negó la condición original.

Iteración controlada numéricamente

La construcción de pseudocódigo

```
ejecuta i = Li, Ls
  e
```

en realidad es una instrucción original de los lenguajes de programación, por lo que merece menos el nombre de pseudocódigo que casi todas las demás.

Cuando el límite inferior es numéricamente menor que el inferior, la iteración se conoce como progresiva (y es el caso usual), mientras que se llama

iteración decreciente en el caso contrario. En Pascal se expresa de dos formas:

Progresiva	Decreciente
`for i := 0 to 25 do`	`for i := 25 downto 0 do`
`begin`	`begin`
`.`	`.`
`.`	`.`
`.`	`.`
`end ;`	`end ;`

La versión progresiva puede ser codificada casi inmediatamente en FORTRAN, cuidando tan sólo que el índice de control (i en este caso) tenga como límite inferior 1 y no 0, e incrementando también entonces el límite superior en la misma proporción (de la misma forma, habrá que tener cuidado con los posibles efectos secundarios que esta alteración acarree, porque en FORTRAN el `DO` *siempre* se ejecuta al menos una vez, aunque desde el inicio la condición de terminación ya esté cumplida):

```
      DO 100 I = 1,26
          .
          .
          .
100   CONTINUE
```

La versión decreciente requiere un truco para codificarse, ya que FORTRAN la prohíbe. Entonces, la idea es utilizar una variable adicional para que desempeñe el papel de índice decreciente.

Este programa imprime los valores (previamente definidos) de un vector en orden decreciente:

```
program ejemplo ;
var LISTA : array [1..10] of real ;
    i : integer ;
begin
.
.
.
for i := 10 downto 1 do
 writeln( LISTA[i] ) ;
.
.
.
end.
```

La codificación en FORTRAN es:

```
      INTEGER AUX
      REAL LISTA(10)

        .
        .
        .
      AUX = 10
      DO 100 I = 1,10
        WRITE(*,*) LISTA(AUX)
        AUX = AUX - 1
100   CONTINUE
        .
        .
        .
      END
```

Selección múltiple

La construcción caso del pseudocódigo es muy útil para expresar algoritmos de manera elegante, y puede usarse directamente en Pascal. Ya no le resultará extraño al lector tener que simularla en FORTRAN.

El siguiente programa muestra en la pantalla de la computadora un menú para que el usuario escoja alguna acción deseada. Supóngase que existen seis módulos ya programados, los cuales efectúan otras tantas funciones que por el momento no nos preocupan:

```
repite
comienza
  escribe  "Digite su selección (1-4)"
  escribe  "Para terminar, digite 0 :"
  lee  dígito
  caso  dígito de
    0: ;
    1: llama  UNO
    2: llama  DOS
    3: llama  TRES
    4: llama  CUATRO
    5: llama  CINCO
     : llama  ERROR
  fin-caso
termina
hasta (dígito = 0)
.
.
.
```

FORTRAN no dispone de la estructura caso. Pascal sí la tiene (**case**), aunque no es completa, ya que no tiene la posibilidad de etiquetas vacías ni maneja adecuadamente el caso cuando la variable de control no toma ninguno de los valores definidos en las etiquetas. Así pues, habrá que simularlas:

```
program seleccion ;
var digito, i : integer ;
begin
repeat
  write( 'Digite su seleccion (1-5)' ) ;
  writeln ;
  write( 'Para terminar, digite 0 :' ) ;
  readln ( digito ) ;
  if digito in [0..5]
    then
      case digito of
        0: ;
        1: i := 1 ;
        2: i := 2 ;
        3: i := 3 ;
        4: i := 4 ;
        5: i := 5 ;
      end
    else ERROR
  write ( i ) ;
  writeln
until digito = 0   ;
end.
```

Hay varios puntos que conviene notar. Pascal no usa la palabra llama, sino que simplemente menciona el nombre del módulo que está siendo llamado; la instrucción **write** no baja el cursor de la pantalla cuando escribe un mensaje, sino que lo mantiene en la última posición, mientras que la instrucción **writeln** sirve exclusivamente para cambiar de renglón.

Se empleó una nueva instrucción de Pascal, bastante útil, llamada **in**, y que sirve para probar si una variable tiene o no un valor dentro de cierto rango. Se tuvo que usar, pues el **case** de Pascal carece, como se dijo, de la etiqueta vacía, que da la posibilidad de atrapar errores.

Este programa debe codificarse en FORTRAN construyendo la estructura por medio de los IF y GOTO. A estas alturas ya debe estar claro para el lector que se está empleando la función "desestructurante" GOTO, pero que se hace bajo la guía constante de las estructuras de control ya aprendidas, lo cual es muy diferente del uso normal y caótico que desafortunadamente suele dársele.

He aquí la codificación:

```
      INTEGER DIGITO
100   CONTINUE
      WRITE(*,*) "DIGITE SU SELECCION (0-5)"
      WRITE(*,*) "PARA TERMINAR, DIGITE 0 :"
      READ(*,*) DIGITO
      IF( DIGITO .EQ. 0 ) GOTO 200
      IF( DIGITO .NE. 1 ) GOTO 110
C                  CASO 1
      CALL UNO
      GOTO 100
110   IF( DIGITO .NE. 2 ) GOTO 120
C                  CASO 2
      CALL DOS
      GOTO 100
120   IF( DIGITO .NE. 3 ) GOTO 130
C                  CASO 3
      CALL TRES
      GOTO 100
130   IF( DIGITO .NE. 4 ) GOTO 140
C                  CASO 4
      CALL CUATRO
      GOTO 100
140   IF( DIGITO .NE. 5 ) GOTO 150
C                  CASO 5
      CALL CINCO
      GOTO 100
C                  CASO DE ERROR, PORQUE DIGITO NO ESTA ENTRE 0 Y 5
150   CALL ERROR
      GOTO 100
C                  FIN DE LA ESTRUCTURA
200   CONTINUE
      END
```

Ésa no es la única manera de codificar el programa, pero sí parece ser la más general. Por otro lado, si se está seguro de que ninguna de las subrutinas `UNO ... CINCO` o `ERROR` cambiará el valor de la variable `DIGITO`, entonces es posible codificarlo así:

```
      INTEGER DIGITO
100   CONTINUE
      WRITE(*,*) "DIGITE SU SELECCION (0-5)"
      WRITE(*,*) "PARA TERMINAR, DIGITE 0 :"
      READ(*,*) DIGITO
      IF( DIGITO .EQ. 0 ) GOTO 200
C                  SIMULACION RESTRINGIDA DEL CASO
      IF( DIGITO .LT. 0  .OR.  DIGITO .GT. 5 ) CALL ERROR
      IF( DIGITO .EQ. 1 ) CALL UNO
      IF( DIGITO .EQ. 2 ) CALL DOS
      IF( DIGITO .EQ. 3 ) CALL TRES
```

```
      IF( DIGITO .EQ. 4 ) CALL CUATRO
      IF( DIGITO .EQ. 5 ) CALL CINCO
      GOTO 100
C                  FIN DE LA ESTRUCTURA
200   CONTINUE
      END
```

Se toma ahora el ejemplo de la pág. 230 (multiplicación de matrices) para codificarlo en los dos lenguajes (la codificación en BASIC aparece en el anexo 7.8):

```
program matmult ;
           (* Multiplicacion de matrices en Pascal *)
    (* declaracion de variables y rangos maximos aceptables *)
const lim = 10 ;
type matriz = array [1..lim,1..lim] of real ;
var i : 1..lim ; j : 1..lim ; k : 1..lim ;
    m : 1..lim ; n : 1..lim ; p : 1..lim ;
    mal : integer ;
    A : matriz ;
    B : matriz ;
    C : matriz ;
begin
repeat
  mal := 1 ;
  writeln ;
  write( 'Dame el numero de renglones de la matriz A (1-10): ' ) ;
  readln( m ) ;
  write( ' Dame el numero de columnas de la matriz A (1-10): ' ) ;
  readln( n ) ;
  write( ' Dame el numero de columnas de la matriz B (1-10): ' ) ;
  readln( p ) ;
  if (m in [1..lim]) and  (n in [1..lim])  and  (p in [1..lim])
     then mal := 0
     else write( 'Hay un error en estos rangos' )
until mal = 0 ;
     (* se leen los valores de las matrices *)
writeln ; writeln ;
for i := 1 to m do
begin
  writeln ;
  for j := 1 to n do
  begin
    write( 'Valor de A[', i, ',', j, ']: ' ) ;
    readln( A[i,j] )
  end
end ;
```

```
writeln ; writeln ;
for i := 1 to n do
begin
  writeln ;
  for j := 1 to p do
  begin
    write( 'Valor de B[', i, ',', j, ']: ' ) ;
    readln(  B[i,j] )
  end
end ;
writeln ; writeln ;
write( 'La matriz producto C[', m, ',', p, '] es: ' ) ;
writeln ; writeln ;
     (* Comienza el calculo *)
for i := 1 to n do
 for j := 1 to p do
 begin
   C[i,j] := 0 ;
   for k := 1 to n do
    C[i,j] := C[i,j] + A[i,k] * B[k,j]
 end ;
     (* Se imprimen los resultados *)
 for i := 1 to m do
 begin
  for j := 1 to p do
   write( C[i,j], '  ' ) ;
  writeln
 end
end.
```

En este programa se volvió a utilizar la instrucción `in` de Pascal para verificar que los rangos sean aceptables. Obsérvese que en la declaración misma de las matrices se usaron otras facilidades de este lenguaje: una que define rangos durante la compilación, como en el caso de

```
var i : 1..lim ;
```

que declara a la variable `i` con rangos aceptables de valores entre 1 y la constante `lim`, y que a su vez fuera declarada por medio de una instrucción anterior; y la otra, que permite definir nuevas variables en términos de otras previamente declaradas. Para esto se requiere el uso de la palabra especial **type**, que define nuevos tipos de variables, en este caso llamadas **matriz**.

La codificación en FORTRAN es bastante natural, y esto se debe a que el lenguaje se presta para trabajos de tipo matemático o numérico:

```
C PROGRAMA PARA MULTIPLICAR MATRICES, ESCRITO EN FORTRAN
      DIMENSION A(10,10), B(10,10), C(10,10)
      INTEGER P
C                  SE LEEN LOS RANGOS Y VALORES DE LAS MATRICES
100   WRITE(*,*)
      WRITE(*,*) "DAME EL NUMERO DE RENGLONES DE LA MATRIZ A:"
      READ(*,*) M
      WRITE(*,*) " DAME EL NUMERO DE COLUMNAS DE LA MATRIZ A:"
      READ(*,*) N
      WRITE(*,*) " DAME EL NUMERO DE COLUMNAS DE LA MATRIZ B:"
      READ(*,*) P
      IF( ( M .GT. 0 .AND.  M .LE. 10 ) .AND.
     $    ( N .GT. 0 .AND.  N .LE. 10 ) .AND.
     $    ( P .GT. 0 .AND.  P .LE. 10 ) ) GOTO 120
      WRITE(*,*) "HAY UN ERROR EN ESTOS RANGOS."
      GOTO 100
120   CONTINUE
      DO 130 I = 1,M
        WRITE(*,*)
        DO 130 J = 1,N
          WRITE(*,*) "VALOR DE A(", I, ",", J, "): "
          READ(*,*) A(I,J)
130   CONTINUE
C
      DO 140 I = 1,N
        WRITE(*,*)
        DO 140 J = 1,P
          WRITE(*,*) "VALOR DE B(", I, ",", J, "): "
          READ(*,*) B(I,J)
140   CONTINUE
C
      WRITE(*,*) "LA MATRIZ PRODUCTO C(", M, ",", P, ") ES : "
      WRITE(*,*)
C                  COMIENZA EL CALCULO
      DO 180 I = 1,M
        DO 180 J = 1,P
          C(I,J) = 0.0
          DO 180 K = 1,N
            C(I,J) = C(I,J) + A(I,K) * B(K,J)
180   CONTINUE
C                  SE IMPRIMEN LOS RESULTADOS
      DO 200 I = 1,M
        WRITE(*,*) ( C(I,J), J = 1,P )
200   CONTINUE
      END
```

Lo único que puede ser novedoso es el renglón que dice

```
WRITE(*,*) ( C(I,J), J = 1,P )
```

que es la manera en que FORTRAN imprime varios valores en un mismo renglón, ya que de otra forma aparecería un número aislado en cada línea, perdiéndose así la apariencia de una matriz. Sugerimos al lector que consulte su manual de FORTRAN para determinar la forma exacta de lograr esto.

8.4 Módulos y subrutinas

En este campo, FORTRAN difiere bastante de Pascal, por lo que nos veremos obligados a discutirlos por separado.

FORTRAN es un lenguaje estático: existe un programa principal y varias (o ninguna) subrutinas independientes que se compilan por separado. Esto significa que, en principio, el programa principal no conoce las subrutinas ni, por tanto, los valores de sus variables. La mayoría de los compiladores permiten que el mismo archivo fuente contenga tanto el programa principal como las subrutinas, que simplemente se escribirán a continuación del primer `END`. Cada una de ellas comienza con la palabra `SUBROUTINE <nombre>`, y termina con la palabra `END`. Además, debe haber al menos una incidencia de la palabra `RETURN`.

(Se emplean los metasignos < y > para delimitar campos con contenido variable. Por ejemplo, `<nombre>` representa alguna palabra que el programador desee emplear para nombrar la subrutina.)

Entonces, existen dos maneras fundamentales de que un programa en FORTRAN entre en contacto con sus subrutinas: por medio del paso de parámetros, y por medio de la proposición `COMMON`. La primera de estas formas se acerca bastante a lo explicado en el capítulo 7 (Sec. 7.4).

La proposición `COMMON` merece tratamiento aparte: su función es, como su nombre en inglés indica, declarar que ciertas variables serán comunes entre el programa principal y todas aquellas subrutinas en las que éste aparezca.

En ese sentido, estos dos programas (que son una variante del expuesto en la pág. 232) son similares, y se obtienen los mismos resultados, aunque aún no se puede decir que sean iguales:

```
C     PROGRAMA PRINCIPAL, QUE USA PASO DE PARAMETROS
      WRITE(*,*) " DAME EL VALOR DEL PRIMER SUMANDO (A):"
      READ(*,*) A
      WRITE(*,*) " DAME EL VALOR DEL SEGUNDO SUMANDO (B):"
      READ(*,*) B
      CALL SUMA( A,B )
      END
C
      SUBROUTINE SUMA( S1,S2 )
      R = S1 + S2
      WRITE(*,*) " A + B = ", R
      RETURN
      END
```

Y esta es la versión equivalente:

```
C     PROGRAMA PRINCIPAL, QUE USA LA PROPOSICION COMMON
      COMMON A,B
      WRITE(*,*) " DAME EL VALOR DEL PRIMER SUMANDO (A):"
      READ(*,*) A
      WRITE(*,*) "DAME EL VALOR DEL SEGUNDO SUMANDO (B):"
      READ(*,*) B
      CALL SUMA
      END
C
      SUBROUTINE SUMA
      COMMON S1,S2
      R = S1 + S2
      WRITE(*,*) " A + B = ", R
      RETURN
      END
```

En la segunda versión, la proposición COMMON eliminó la necesidad del paso de los parámetros, ya que logra que el programa principal y la subrutina tengan acceso a las mismas variables (que están declaradas por medio de éste).

Es importante hacer notar que el COMMON comparte las variables estrictamente en el orden en que están declaradas y que, por tanto, no es el nombre de cada una de ellas el factor dominante, sino su posición relativa. Está claro, entonces, que la variable B del programa principal es la misma que S2 de la subrutina, porque en las dos declaraciones COMMON ocupa el segundo lugar.

Efectos secundarios de la proposición COMMON

Aparentemente, las dos formas de manejar las variables entre rutinas son equivalentes, pero esto no es del todo cierto. Cuando un módulo llama a otro y le pasa una lista de parámetros, son sólo esas variables las que pueden resultar afectadas por cualquier acción que se realice sobre ellas (incrementarlas, cambiarlas de valor, etc.), pero cuando se usa COMMON, todas las variables allí declaradas se vuelven comunes, estando así más expuestas a algún cambio de valor no previsto.

El estudio detallado de estos posibles efectos secundarios constituye por sí mismo un tema específico del diseño de sistemas computacionales, donde se manejan conceptos como **cohesión** y **acoplamiento** de sistemas, que no tenemos oportunidad de tratar en este nivel. Este tema, y otros relacionados, conforman la materia de estudio de una disciplina especializada, conocida como **ingeniería de software**. Las referencias [FAIR87], [SOMI88] y [YOUE79] son ejemplos de textos dedicados a esto.

Si se hubiera usado la declaración

```
COMMON A, B, RESULT
```

(y su correspondiente COMMON S1, S2, R en la subrutina) en lugar de la que se empleó, entonces la variable RESULT sería propiedad tanto del programa

principal como de la subrutina que lo obtiene, siendo que, en realidad, es tan sólo esta última la que debería ser capaz de hacerle algún cambio.

Efectos de este tipo pueden ser potencialmente peligrosos en un sistema que maneja varias decenas de subrutinas, por lo que se recomienda hacer un uso cuidadoso del COMMON.

Pascal, por otro lado, es un lenguaje que emplea una filosofía diferente para el manejo de la modularidad. En términos generales podría decirse que usa un esquema jerárquico para el paso de parámetros y para la transmisión de la información que manejan las variables.

El concepto de subrutina independiente no se utiliza en Pascal más que de manera especial, siendo genérico el uso de los llamados **procedure**, (o módulos) que forman parte misma del texto del programa principal.

Es decir, un programa en Pascal consiste, como ya se vio, en un texto que comienza con la palabra especial **program**, y que termina con la palabra **end.**, donde el punto final es obligatorio. Si se desea incluir módulos, éstos aparecerán en algún punto intermedio del programa, y deberán comenzar con la palabra **procedure**, seguida del nombre del módulo y del punto y coma. Todo **procedure** termina con su correspondiente **end**; (con punto y coma), que "cierra" el **begin** que necesariamente debe aparecer (no existe la instrucción RETURN como en FORTRAN).

Existe otro tipo de módulos en Pascal, llamados **function** (que devuelven un solo valor), y que son similares al FUNCTION de FORTRAN, que se estudia más adelante.

Estructuras de bloques en Pascal

Es importante tomar en cuenta que los módulos de Pascal *no son ejecutables*, sino tan sólo *invocables*, por medio de su nombre. Esto quiere decir que el flujo de control dentro de un programa con estructura de bloques de este tipo delimita cualquier número de renglones que comience con la palabra **procedure**, y sólo ejecuta las instrucciones del programa principal (que por supuesto podrán contener llamadas a los módulos), mismas que están entre el primer **begin** y el **end.** final (con punto).

La buena práctica de la programación en lenguajes con estructura de bloques (como Pascal, Algol, PL/I, etc.) dicta que los módulos se escriban todos agrupados al inicio (o al final) del programa principal, para dejar claramente diferenciada el área ejecutable del área que puede ser llamada.

Como caso particular, en Pascal los **procedure** deben escribirse al comienzo del programa principal, antes de su **begin**.

La estructura que se propone, entonces, es de la siguiente forma para un programa principal y dos módulos (se dejaron renglones en blanco para mayor claridad):

```
program principal ;
     (* aquí van las declaraciones de los datos del programa principal
          aunque no necesariamente todas las que aquí aparecen *)
const    ....  ;
type     ....  ;
var      ....  ;

procedure modulo1 ;       (* inicia el texto del primer módulo *)
     const    ....  ;    (* con sus declaraciones de datos locales *)
     type     ....  ;          (* si es que se requieren *)
     var      ....  ;
     begin      (* inicio de las instrucciones del primer módulo *)
       .
       .
       .
     end ;      (* termina el primer módulo *)
procedure modulo2 ;       (* inicia el texto del segundo módulo *)
     const    ....  ;    (* con sus declaraciones de datos locales *)
     type     ....  ;          (* si es que se requieren *)
     var      ....  ;
     begin      (* inicio de las instrucciones del segundo módulo *)
       .
       .
       .
     end ;      (* termina el segundo módulo *)

begin    (* aquí comienzan las instrucciones ejecutables *)
  .    (* del programa principal, únicas que son directamente *)
  .                    (* ejecutables *)
  .
  .
  modulo2 ;  (* ésta es una llamada *)
  .
  .
  .
  modulo1 ;  (* ésta es otra *)
  .
  .
  .

end.      (* aquí termina el programa principal y, con él,
                                todo el programa completo *)
```

Surge una pregunta: ¿cómo puede el programa principal pasar (o recibir) información hacia (o desde) un módulo? Existen dos formas: mediante parámetros, y por medio de lo que llamaremos la regla general del manejo de bloques.

El primer caso es parecido al de FORTRAN, pero depende también en cierto grado de la regla general, por lo que ésta se explica primero.

Regla general del manejo de bloques: Todas las variables declaradas en el programa principal son pasadas a los módulos (bloques) internos como "herencia", a menos que éstos decidan rechazarlas declarando sus propias variables localmente.

Para fines de comparación, lo que la regla dice es que existe de forma implícita una especie de "super-COMMON" entre el programa principal y todos sus módulos internos. Sin embargo, si uno de ellos requiere la utilización de variables que no sean globales, entonces puede definirlas localmente, en el entendido de que éstas no serán accesibles desde ningún punto que esté fuera de ese módulo. Como tal vez esté claro ya, este mecanismo evita los efectos secundarios que se mencionaron en el caso de FORTRAN, puesto que sólo el módulo en cuestión puede alterar los valores de sus variables locales.

Esta regla es recursiva en el sentido de que, si se desea, un módulo cualquiera puede tomar el papel del programa principal y englobar otros, pasándoles a ellos su herencia, en la forma descrita; estos módulos de tercer nivel podrán rechazarla, declarando sus propias variables, y así sucesivamente.

Se establece así una jerarquía de herencias, que en la literatura en inglés se conoce como *the scope of variables* (el alcance de las variables). Como en el caso de las estructuras de control, conviene no abusar del anidamiento de módulos, porque no aporta mucho a la claridad de los programas.

El programa principal es el padre (siguiendo con la idea de la jerarquía familiar) de todos los módulos, y uno puede convertirse en padre de otro si lo engloba físicamente. Dos **procedure** pueden ser hermanos si ambos son hijos del **program** principal y ninguno contiene al otro.

De la misma forma, los hijos de un **procedure** cualquiera solamente pueden ser llamados por él mismo o por su padre. Esto significa que, al igual que con las variables, los módulos en Pascal pueden ser o no compartidos. Todo esto puede parecer confuso a primera vista, pero no es más que un resultado de la regla general (recursiva) de manejo de bloques.

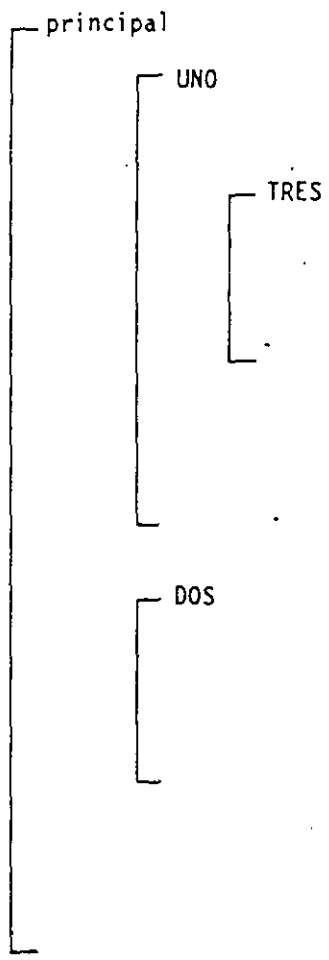
En esta estructura, los módulos UNO y DOS son hermanos, y el módulo TRES es hijo de UNO, por lo que DOS no puede llamarlo.

```
program principal ;
   var  alfa, beta : integer ;

procedure UNO ;
   var  zeta : real ;

   procedure TRES ;
        begin
          .
          .
          .
        end  (* TRES *)
      begin  (* UNO *)
        .
        .
        .
      end ;  (* UNO *)

    procedure DOS ;
      var beta : char ;
      begin
        .
        .
        .
      end ;  (* DOS *)
      begin  (* principal *)
        .
        .
        .
      end.   (* principal *)
```



Además, la herencia que el programa principal pasa a todos sus hijos (las variables enteras **alfa** y **beta**) es rechazada en parte por el módulo DOS, porque éste vuelve a declarar la variable **beta** como de tipo de caracteres.

Igualmente, el hijo de UNO (o sea, el módulo TRES) puede usar la variable real **zeta**, que le fue pasada por su padre, además de las enteras **alfa** y **beta**.

He aquí la codificación equivalente en Pascal del programa en FORTRAN con COMMON que usa un módulo para hacer una suma:

```
program principal ;
var A, B : real ;
procedure SUMA ;
   var R : real ;  (* variable local a SUMA *)
   begin
     R := A + B ;
    writeln( 'A + B = ', R )
  end ;   (* SUMA *)
  begin   (* principal *)
  write( ' DAME EL VALOR DEL PRIMER SUMANDO (A):' ) ;
  readln( A ) ;
  write( 'DAME EL VALOR DEL SEGUNDO SUMANDO (B):' ) ;
  readln( B ) ;
  SUMA
  end.
```

Antes de pasar a analizar el manejo de parámetros en Pascal, pedimos al lector que dedique unos minutos a entender el funcionamiento de un nuevo programa (que ciertamente es un poco rebuscado, pero representa una buena oportunidad para afirmar los conceptos recién explicados).

Este es un ejemplo de la ejecución y de los valores que entrega (aparecen subrayados los valores que se dieron como respuesta a las peticiones):

```
DAME EL VALOR REAL (A): 1.0
DAME EL VALOR REAL (B): 2.0
DAME EL VALOR ENTERO (J): 4
DAME EL VALOR ENTERO (K): 5
A + B =  3.00000

J + K = 9

A = 1.00000
B = 2.00000
A + B =  1.10412E-05
```

Este último número es la respuesta que dio nuestra computadora cuando se le pidió que imprimiera el valor de la variable R, local al módulo DOS; debe ser considerado como "basura" (pudo haber sido cero, o cualquier otro, porque estaba declarado pero nunca se le asignó ningún valor).

He aquí el programa:

```
program principal ;
var A, B : real ;
procedure UNO  ;
var J, K : integer ;
  procedure TRES ;
   var R : real ;  (* variables locales a TRES *)
       I : integer ;
   begin   (* TRES *)
     R := A + B ;
     writeln( 'A + B = ', R ) ;
     writeln ;
     I := J + K ;
     writeln( 'J + K = ', I )
     writeln ;
   end ;   (* TRES *)
   begin   (* UNO *)
     write( ' DAME EL VALOR ENTERO (J):' ) ;
     readln( J ) ;
     write( ' DAME EL VALOR ENTERO (K):' ) ;
     readln( K ) ;
     TRES
   end ;   (* UNO *)
   procedure DOS ;
    var R : real ;  (* Esta no es la misma "R" que en el modulo TRES *)
    begin  (* DOS *)
      writeln( 'A = ', A  ) ;
      writeln( 'B = ', B  ) ;
      writeln( 'A + B = ', R )
    end;   (* DOS *)
   begin   (* principal *)
   write( ' DAME EL VALOR REAL (A):' ) ;
   readln( A ) ;
   write( ' DAME EL VALOR REAL (B):' ) ;
   readln( B ) ;
   UNO ;
   DOS
   end.
```

Como se dijo anteriormente, el manejo de los parámetros en Pascal está supeditado a la regla general de alcance de las variables. En términos generales, se emplea para lograr que un mismo procedure trabaje sobre diferentes conjuntos de datos cuando es llamado en diferentes ocasiones. Una llamada a un módulo al que se pasa una lista de argumentos aparece así:

```
DIVIDE ( 355, 113 ) ;
```

y la definición del módulo dentro del programa principal es, por ejemplo:

```
procedure  DIVIDE ( x : real; y : real ) ;
begin
  writeln ( x, ' entre ', y, ' = ', x/y )
end ;
```

Aquí, en la lista de parámetros formales del **procedure** se exige que estén declarados (separados con punto y coma si son más de uno) con los mismos atributos que se espera de ellos a la hora de la invocación.

Dentro del mismo programa, esta es otra llamada válida:

```
DIVIDE( 0.15, -1.14) ;
```

Es importante tomar en cuenta que los parámetros son estrictamente locales al módulo donde se declaran y que, por tanto, cualquier referencia a ellos fuera del módulo en cuestión sería ilegal, dando lugar a un mensaje de error por parte del compilador. Esto tiene sus ventajas, ya que un módulo no puede cambiar el valor original de ninguna variable que le fuera pasada como parámetro por su padre, sino que puede hacer uso de ella sin temer efecto secundario alguno.

Supóngase que existe un módulo especial para invertir matrices (algunos métodos matemáticos eficientes para lograr esto tienen como desventaja el hecho de que destruyen los valores originales de la matriz, siendo entonces peligroso su uso indiscriminado). Si la matriz pasa al módulo como parámetro, entonces el programa original no tiene nada que temer, pues ningún cambio que sufra se verá reflejado en el programa principal. Esto significa, necesariamente, que el compilador de Pascal generó código para que los valores de estos parámetros sean copiados por el módulo invocado (para que en realidad se trabaje con una copia de la matriz, y no con la original). Si bien este esquema, denominado **paso de parámetros por valor**, es seguro, también es costoso, por lo que se recomienda mesura al usarlo. (Algunos compiladores de Pascal no permiten el paso de arreglos por valor.)

Tipos de paso de parámetros

Cuando se requiere que tanto el módulo como el programa principal trabajen físicamente sobre el mismo parámetro (y no uno con el original y el otro con la copia), entonces hay que declararlo como **var** (variable) dentro del **procedure**. Si esto sucede, entonces todos los cambios de valor que el módulo haga a ese parámetro serán permanentes, y tendrán repercusiones en el programa principal. Este esquema se conoce como **paso de parámetros por referencia** y, además, es el único que se usa en FORTRAN.

En ambos esquemas, si se desea pasar un arreglo (**array**) como parámetro, tendrá que haber sido declarado dentro de una especificación **type**, puesto que Pascal no permite declaraciones complejas dentro de la lista de parámetros formales. Por ejemplo, esta declaración es *incorrecta*:

```
procedure EQUIS ( alfa : array [1..25] of integer ) ;
```

La forma correcta de hacerlo es, como se dijo, empleando la cláusula type:

```
type vector = array [1..25] of integer ;
   .
   .
   .
procedure EQUIS ( alfa : vector ) ;
```

El siguiente programa es un ejemplo muy sencillo del uso de los dos tipos de parámetros ya explicados, donde la variable i se pasa por referencia y la variable j se pasa por valor, siendo tan sólo la primera afectada por los cambios que le hizo el módulo:

```
programa principal ;
var i, j : integer ;
  procedure UNO ( var i : integer ; j : integer ) ;
                  (* Observese el paso del parametro por referencia *)
  begin
    i := i + 1 ;
    j := j - 1 ;
    writeln ( 'UNO: para mi, -i- vale ', i, ' y -j- vale ', j ) ;
    writeln
  end ; (* UNO *)
begin (* principal *)
  i := 20 ;
  j := 90 ;
  writeln ( 'principal: originalmente, -i- vale ", i, ' y -j- vale ', j ) ;
  writeln ;
  UNO ( i, j ) ;
  writeln ( 'principal: y ahora, -i- vale ', i, ' y -j- vale ', j )
end.   (* principal *)
```

Examínelo el lector con cuidado para entender que si entrega estos resultados:

```
principal: originalmente, -i- vale 20 y -j- vale 90

UNO: para mí, -i- vale 21 y -j- vale 89

principal: y ahora, -i- vale 21 y -j- vale 90
```

es porque, desde el punto de vista del programa principal, la variable i sí es afectada por el módulo (porque fue declarada como parámetro variable), mientras que j sólo es alterada de forma local.

En resumen, desde el punto de vista del programa principal (o de cualquier módulo padre), cuando se desea que un parámetro sirva solamente para llevar valores, se empleará la declaración por valor (*i.e., sin* la palabra var) mientras que, cuando un parámetro deba traer valores de regreso, deberá emplearse la declaración var (por referencia).

Funciones

El manejo de funciones en Pascal y FORTRAN es similar, y se emplea cuando todo lo que se espera de un módulo es un solo valor como resultado. Como ya debe estar claro, un `procedure` o una `SUBROUTINE` tienen la capacidad de calcular, y devolver al programa que las llamó, varios valores, que corresponden a las variables que hayan sido declaradas como parámetros.

Sin embargo, muchas veces es deseable diseñar un módulo que se comporte como si fuera una instrucción directamente ejecutable del lenguaje de programación, que sirve para evaluar cierta función específica.

Supóngase que en la especificación de un sistema se determina la necesidad de leer un vector y encontrar el valor más grande que contenga. El módulo encargado de esto sólo tiene que conocer el vector y procesarlo de tal forma que encuentre el resultado pedido. Podría codificarse en forma de `procedure` o `SUBROUTINE`, pero también es un buen candidato para formar una función, porque se espera que entregue un solo valor final.

Como ventaja adicional se tiene que ahora el sistema parece disponer de una nueva instrucción, que sirve exactamente para los fines deseados.

El algoritmo consiste en suponer que el primer elemento del vector es el número mayor, e ir examinando secuencialmente todos los demás valores hasta demostrar lo contrario. Si se encuentra un número mayor, simplemente se adopta como si fuera el máximo y se repite el mismo esquema hasta agotar el vector.

Su diseño en pseudocódigo es:

```
! leer un vector y obtener el número más grande.
! se supone que se tiene acceso al vector en cuestión.
máximo = vector(1)
índice = 2
mientras ( no se termine el vector )
comienza
   si vector(índice) > máximo entonces máximo = vector(índice)
   índice = índice + 1
termina
escribe máximo
```

En FORTRAN el programa se escribiría así:

```
      INTEGER ALFA(10)
      INTEGER BUSCAM
      DO 100 I = 1,10
         WRITE(*,*) "DAME EL VALOR NO. ", I
         READ(*,*) ALFA(I)
100   CONTINUE
      MAXIMO = BUSCAM ( ALFA )
      WRITE(*,*) "EL VALOR MAS GRANDE DEL VECTOR FUE", MAXIMO
      END
C
      INTEGER FUNCTION BUSCAM( VECTOR )
      INTEGER VECTOR(10)
      MAXIMO = VECTOR(1)
      INDICE = 2
C        ESTRUCTURA TIPO 'MIENTRAS'
100   CONTINUE
      IF( INDICE .GT. 10 ) GOTO 200
        IF( VECTOR(INDICE) .GT. MAXIMO ) MAXIMO = VECTOR(INDICE)
        INDICE = INDICE + 1
        GOTO 100
200   CONTINUE
      BUSCAM = MAXIMO
      RETURN
      END
```

Hay que tener cuidado de que la definición del tipo de la función (`INTEGER FUNCTION` en este caso) corresponda con lo dicho en el programa principal. Como en FORTRAN toda variable que comienza con la letra `B` es real (a menos que se diga lo contrario), fue necesario declarar

```
INTEGER BUSCAM
```

en el programa principal, para que concordaran los tipos.

Como en todo programa en FORTRAN, el programa principal y las subrutinas o funciones se compilan por separado, razón por la cual las variables y etiquetas de uno no tienen nada que ver con las de los demás y entonces pueden emplearse en forma independiente.

Para que este programa fuera más parecido al de Pascal hubiéramos deseado escribir

```
WRITE(*,*) "EL VALOR MAS GRANDE DEL VECTOR FUE", BUSCAM( ALFA )
```

pero la sintaxis de FORTRAN lo prohíbe, por lo que se escribió el renglón que dice `MAXIMO = BUSCAM( ALFA )` antes de escribir el resultado.

Por último, la variable `VECTOR` desempeña el papel de parámetro formal (o ficticio), que recibe por referencia los valores reales del arreglo `ALFA` del programa principal. Como en cualquier paso de parámetros por referencia (parámetros `var` en Pascal), hay que tener cuidado con los posibles efectos secundarios.

Por otro lado, la codificación en Pascal es inmediata:

```
program principal ;
const lim = 10 ;
type vector = array [1..lim] of integer ;
var i : integer ;
    alfa : vector ;
     function buscam ( var alfa : vector ) : integer ;
     var maximo, indice : integer ;
     begin
       maximo := alfa[1] ;
       indice := 2 ;
       while indice <= lim do
       begin
         if alfa[indice] > maximo then maximo := alfa[indice] ;
         indice := indice + 1
       end ;
       buscam := maximo
     end ;  (* buscam *)
begin (* principal *)
  for i:= 1 to lim do
  begin
    write( 'Dame el valor No. ', i, ' :' ) ;
    readln( alfa[i] )
  end ;
  writeln( 'El valor mas grande del vector fue ', buscam( alfa ) )
end.  (* principal *)
```

Lo importante aquí es tomar nota de la declaración de una función:

`function` <nombre> (<lista de parámetros>) : <tipo>

donde <tipo> es la descripción del único valor que devolverá la función en cuestión. Por fuerza, al menos una vez debe aparecer una instrucción de la forma

<nombre> := <expresión>

dentro de la definición de la función, que servirá para devolver el valor obtenido, que debe ser precisamente del tipo declarado.

La llamada de la función simplemente consiste en mencionar su nombre seguido de los argumentos necesarios, como si fuera una variable simple. En el programa de ejemplo, esto se hizo dentro de la instrucción `writeln`, en el último renglón ejecutable.

Decidimos usar un parámetro `var` (por referencia) para evitar que la función tenga que copiar los valores del vector. Antes se advirtió al lector de los posibles efectos secundarios de este tipo de paso de parámetros, aunque en este programa en particular los riesgos son inexistentes.

8.5 Ejemplo de un diseño completo codificado

A continuación se expone y comenta la codificación en FORTRAN y Pascal del programa de las ocho damas, que se diseñó en el capítulo anterior.

```
C        PROBLEMA DE LAS OCHO DAMAS CODIFICADO EN FORTRAN
      INTEGER COL, REN, RESULT, ULT
      INTEGER BAJA, SOL, SUBE, VECTOR
      COMMON BAJA(8), SOL, SUBE(8), VECTOR(8)
C        NUMERO DE SOLUCIONES QUE SE DESEA IMPRIMIR .
      DATA  NUMERO  / 7 /
      ULT = 0
      RESULT = 0
      REN = 1
      COL = 1
      SOL = 0
C        COMIENZA LA BUSQUEDA DE SOLUCIONES
      CALL COLOCA( REN,COL )
C        ITERACION TIPO -REPITE-
100   CONTINUE
C                  ITERACION TIPO -MIENTRAS-
110     CONTINUE
        IF( COL .GE. 8 ) GOTO 150
          IF( RESULT .NE. 0 ) GOTO 130
C                  YA SE ENCONTRO UNA SOLUCION, PREPARARSE PARA LA SIGUIE
            COL = COL + 1
            REN = 0
130       CONTINUE
C                  VERIFICAR SI HAY LUGAR LIBRE PARA LA NUEVA DAMA
          CALL LIBRE( REN, COL, RESULT )
          IF( RESULT .EQ. 1 ) CALL ATRAS( REN, COL, ULT )
          IF( RESULT .NE. 1 ) CALL COLOCA( REN, COL )
C                  TERMINA EL -MIENTRAS-
        GOTO 110
150     CONTINUE
C                  SE ENCONTRO UNA SOLUCION: IMPRIMIRLA Y BUSCAR OTRA
        CALL IMPRIM
        CALL ATRAS( REN, COL, ULT )
        RESULT = 1
C                  TERMINA EL -REPITE-
      IF( ULT .EQ. 0  .AND.  SOL .LT. NUMERO ) GOTO 100
      END
```

Como puede observarse, el programa es bastante sencillo, y no hace sino simular, de las formas que ya se han explicado, las estructuras de control propias del diseño original en pseudocódigo de las págs. 239-241.

La variable NUMERO está declarada en un DATA, que es la forma en que FORTRAN permite darle valores iniciales (durante la compilación).

La declaración COMMON incluye tres arreglos, que serán usados por las siguientes subrutinas para determinar si una dama es atacada tanto en la diagonal ascendente o descendente como en el renglón mismo en que se intenta colocarla.

```
      SUBROUTINE COLOCA( REN, COL )
C                  MODULO PARA COLOCAR UNA DAMA EN EL TABLERO
      INTEGER COL, REN, RESULT, ULT
      INTEGER BAJA, SOL, SUBE, VECTOR
      COMMON BAJA(8), SOL, SUBE(8), VECTOR(8)
      VECTOR(COL) = REN
      SUBE(COL) = REN - COL
      BAJA(COL) = REN + COL
      RETURN
      END
```

Esta rutina coloca la dama que se le indica (la COL-ésima) en la posición REN-ésima dentro de su columna. En el arreglo VECTOR guarda la posición en que quedó; en SUBE y BAJA guarda las restas y sumas, respectivamente, de las posiciones en que se colocó la dama, con vistas a facilitar las funciones de la rutina LIBRE.

```
      SUBROUTINE LIBRE( REN, COL, RESULT )
C                 MODULO QUE AVERIGUA SI HAY UNA POSICION LIBRE EN
C                                      LA COLUMNA ACTUAL DE LA DAMA.
      INTEGER COL, REN, RESULT, ULT
      INTEGER BAJA, SOL, SUBE, VECTOR
      COMMON BAJA(8), SOL, SUBE(8), VECTOR(8)
C                 ITERACION TIPO -MIENTRAS-
100   CONTINUE
      IF( REN .GE. 8 ) GOTO 200
        RESULT = 0
        REN = REN + 1
        ICOL = COL - 1
        DO 150 J = 1,ICOL
          IF( .NOT. ( SUBE(J) .EQ. ( REN - COL )  .OR.
     $                BAJA(J) .EQ. ( REN + COL )  .OR.
     $              VECTOR(J) .EQ.   REN        ) ) GOTO 150
          RESULT = 1
          GOTO 160
C                 TERMINAN  EL -IF- Y EL -DO-
150     CONTINUE
C
160     CONTINUE
      IF( RESULT .EQ. 0 ) GOTO 200
C                 TERMINA EL -MIENTRAS-
      GOTO 100
200   CONTINUE
      RETURN
      END
```

Como el pseudocódigo de la rutina no es trivial (contiene varias estructuras anidadas, expuestas en la pág. 240). Su codificación en FORTRAN no tiene toda la claridad deseable, pero aun así no resulta demasiado oscura.

La proposición DO que controla las preguntas por las diagonales es necesaria para asegurarse de que se han averiguado las posibilidades de ataque de *todas* las columnas anteriores a aquélla donde se intenta colocar la nueva dama.

```
      SUBROUTINE ATRAS( REN, COL, ULT )
C                  MODULO PARA HACER 'BACKTRACK'
      INTEGER COL, REN, RESULT, ULT
      INTEGER BAJA, SOL, SUBE, VECTOR
      COMMON BAJA(8), SOL, SUBE(8), VECTOR(8)
      COL = COL - 1
      IF( COL .LT. 1 ) GOTO 110
C                  CLAUSULA -THEN-
        REN = VECTOR(COL)
        VECTOR(COL) = 0
        GOTO 120
C                  CLAUSULA -ELSE-
110   CONTINUE
      ULT = 1
120   CONTINUE
      RETURN
      END
```

Tal como dice su pseudocódigo, esta rutina elimina la dama actual del tablero, porque se demostró que su posición no era buena. Hay que tomar nota del renglón en que se quedó dentro de su columna, para no volver a comenzar desde el primero otra vez, sino seguirla avanzando a partir de ahí.

```
      SUBROUTINE IMPRIM
C                  MODULO PARA IMPRIMIR UNA SOLUCION
      INTEGER COL, REN, RESULT, ULT
      INTEGER BAJA, SOL, SUBE, VECTOR
      INTEGER TAB(8,8)
      COMMON BAJA(8), SOL, SUBE(8), VECTOR(8)
      DO 110 I = 1,8
        DO 110 J = 1,8
          TAB(I,J) = 0
110   CONTINUE
C                  CAMBIO DE PAGINA CADA TRES TABLEROS IMPRESOS
      IF( MOD(SOL,3) .EQ. 0 ) WRITE(*,*) ""
      SOL = SOL + 1
      DO 120 I = 1,8
        TAB( VECTOR(I),I ) = I
120   CONTINUE
      WRITE(*,*)   " "
      WRITE(*,*)
```

```
      WRITE(*,*) "       SOL. NUM. ", SOL
      WRITE(*,*)
      WRITE(*,*)
      DO 130 I = 1,8
        WRITE(*,9) ( TAB(I,J), J = 1,8 )
130   CONTINUE
      DO 140 I = 1,8
        TAB( VECTOR(I),I ) = 0
140   CONTINUE
  9   FORMAT(8I4)
      RETURN
      END
```

Esta es la rutina más complicada, desde el punto de vista de FORTRAN, porque emplea ciertas características de los formatos de este lenguaje. Esta es la razón de la construcción que se conoce como "DO implícito":

```
WRITE(*,9) ( TAB(I,J), J = 1,8 )
```

que ya fue usada anteriormente en este capítulo, en el programa para multiplicar matrices.

El renglón

```
IF( MOD(SOL,3) .EQ. 0 ) WRITE(*,*) "" '
```

hace uso de la operación módulo para pedir un cambio de página cada 3 tableros impresos. Esta operación calcula el residuo de dividir SOL entre 3: si es cero, quiere decir que el valor de SOL es múltiplo exacto de 3 y, por tanto, hay que cambiar de hoja.

En FORTRAN el cambio de hoja suele hacerse por medio de una instrucción WRITE que "imprime" un 1 en la primera posición del renglón: esta es la manera en que se indica a la impresora (que se identifica por medio de la unidad 6) que debe cambiarse al inicio de la nueva página. Aunque en este programa no se usan, las instrucciones para lograr esto son:

```
      IF( MOD(SOL,3) .EQ. 0 ) WRITE(6,20)
20    FORMAT(1H1)
```

Recomendamos la consulta del manual de FORTRAN particular de la computadora en la que se trabaje, porque este concepto ha tenido algunos cambios durante la ya larga historia del lenguaje.

Aquí está el mismo problema, codificado en el lenguaje Pascal. Se han dejado renglones en blanco entre los diversos módulos, por razones de claridad:

```
(*   Problema de las ocho damas, codificado en Pascal  *)
program damas ;
const numero =  7 ;
type arreglo = array [0..7] of integer ;
var baja, sube, vector : arreglo ;
    col, ren, result, sol, ult : integer ;
                  (* Modulo que coloca una dama en el tablero *)
    procedure coloca( var ren, col : integer ) ;
    begin
      vector[col] := ren ;
      sube[col]   := ren - col ;
      baja[col]   := ren + col
    end ;  (* coloca *)

                  (* Modulo que averigua si existe una posicion libre *
    procedure libre( var ren, col, result : integer ) ;
    var j : integer ;
    begin
      result := 1 ;
      while  (ren < 7) and (result <> 0) do
      begin
        result := 0 ;
        ren := ren + 1 ;
        j := 0 ;
        repeat
          if     (sube[j] = ( ren - col ))  or
                 (baja[j] = ( ren + col ))  or
               (vector[j] =   ren           )  then result := 1 ;
          j := j + 1 ;
        until (j >= col) or (result = 1)
      end
    end ;  (* libre *)

                  (* Modulo que regresa todo a la posicion anterior *)
    procedure atras( var ren, col, ult : integer ) ;
    begin
      col := col - 1 ;
      if col >= 0 then begin
                         ren := vector[col] ;
                         vector[col] := 0
                       end
                  else ult := 1
    end ;  (* atras *)
```

```
                    (* Modulo que imprime una solucion *)
    procedure imprime ;
    var tab : array [0..7,0..7] of integer ;
        i, j : integer ;
    begin
      for i := 0 to 7 do
        for j := 0 to 7 do
          tab[i,j] := 0 ;
      if sol mod 3 = 0 then writeln( '' ) ;
      sol := sol + 1 ;
      for i := 0 to 7 do
        tab[ vector[i], i ] := i + 1 ;
      writeln('   Sol.  Num.  ', sol ) ;
      writeln ;
      for i := 0 to 7 do
      begin
        for j := 0 to 7 do
          write( tab[i,j], ' ' ) ;
        writeln
      end ;
      writeln ; writeln ;
      for i := 0 to 7 do
        tab[ vector[i], i ] := 0
    end;

begin   (* programa principal *)
  ult := 0 ; result := 0 ;
  ren := 0 ; col := 0 ; sol := 0 ;
  coloca( ren, col ) ;   (* comienza la busqueda de soluciones *)
  repeat
    while( col < 7 ) do  (* trata de colocar la proxima dama *)
    begin
      if result = 0 then
      begin
        col := col + 1 ; (* ya se encontro una solucion, prepararse *)
        ren := -1        (* para la siguiente. *)
      end ;
                    (* verificar si hay lugar libre para la nueva dama *)

      libre( ren, col, result ) ;
      if result = 1 then atras( ren, col, ult )   (* regresar a la ant. *)
                    else coloca( ren, col )
    end ;
                    (* se encontro una solucion: imprimirla *)
    imprime ;
    atras( ren, col, ult ) ;
    result := 1
  until  (ult <> 0) or (sol >= numero)
end.
```

Como era de esperarse, la codificación en Pascal es más cercana al pseudocódigo original que la de FORTRAN, por lo que no hay mucho que comentar. En la rutina de impresión se hizo uso de las facilidades de Pascal para imprimir varias cosas en un mismo renglón usando la instrucción `write` en lugar de `writeln`. Esto evita tener que usar trucos como el `DO` implícito de FORTRAN.

En Pascal los índices de los arreglos pueden comenzar a partir de cero, y no necesariamente de uno, como en FORTRAN. Obsérvese cómo en algunos módulos se hizo uso de los parámetros por referencia (declarados como `var` en el encabezado de `procedure`), porque sí nos interesa que los cambios que sufran sean permanentes, lo cual no se lograría si no se hubieran declarado así.

Estudie el lector estas codificaciones, y compárelas contra los pseudocódigos originales. Es importante que los codifique y ejecute en alguna computadora a la que tenga acceso.

En todo caso hay que tomar en cuenta que esta no es de ninguna manera "la mejor" codificación para este problema, y que puede perfectamente haber otros esquemas igualmente válidos, cosa que además conviene siempre tener en consideración en el oficio de la programación.

8.6 Manejo de archivos

Por **manejo de archivos** se entiende el uso, desde un lenguaje de programación, de los recursos que ofrece el sistema operativo para almacenar y recuperar información en medios magnéticos externos a la memoria central.

Está claro que la desventaja principal del uso de memoria secundaria con respecto a la central es su velocidad, que es menor en varios órdenes de magnitud. La ventaja, por otro lado, es la gran capacidad que ofrecen los discos y cintas magnéticas comparados con la de la memoria, que es relativamente pequeña.

La cantidad de datos e información que maneja casi cualquier programa o sistema grande excede con mucho el tamaño de la memoria central de la computadora, por lo que el manejo de archivos externos se vuelve indispensable.

Todos los lenguajes de programación consideran, de alguna u otra manera, la creación, uso y mantenimiento de archivos externos. Como se mencionó en el capítulo 4, el sistema operativo se encarga de facilitar los aspectos técnicos y físicos del manejo de periféricos, quedando como única responsabilidad del programador hacer uso eficiente de las instrucciones y operaciones elementales del manejo de archivos.

Estas operaciones son, a grandes rasgos, las siguientes:

- Crear un nuevo archivo.
- Eliminar un archivo ya existente.
- Leer información de un archivo ya existente.

- Escribir información en un nuevo archivo.
- Añadir o cambiar información en un archivo ya existente.

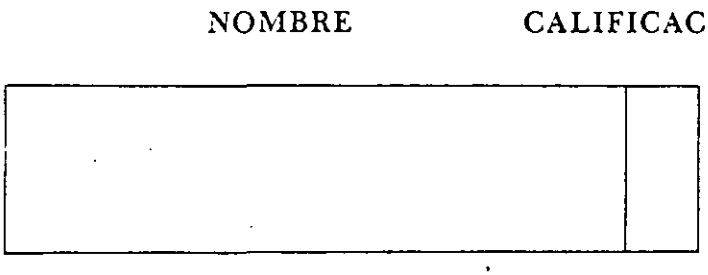
Las cuatro primeras son operaciones elementales que cualquier sistema de manejo de archivos logra directamente. La última, sin embargo, puede llegar a ser sumamente compleja, debido a que existen múltiples maneras de lograrla. Ni Pascal ni FORTRAN incluyen maneras elaboradas de llevar a la práctica esta función, por lo que aquí sólo se hará una descripción muy elemental de ella. No puede ser de otra manera, ya que la cantidad de material e información sobre este tema es tan amplia que fácilmente justificaría todo un texto exclusivo que, por otra parte, sería de carácter más avanzado que éste.

Se propondrá ahora un problema sencillo, que será empleado en todo lo que resta de este capítulo para exponer estos nuevos conceptos y operaciones, junto con la definición de las instrucciones adecuadas de Pascal y FORTRAN para lograrlos.

Supóngase que existe un archivo en disco que contiene un número indeterminado de nombres y calificaciones de alumnos. Hay que diseñar un programa para leer estos datos, imprimirlos y calcular el promedio de las calificaciones. Este pequeño sistema debe permitir también la inclusión, modificación y eliminación de datos.

La fuente fundamental de información será un archivo principal que contendrá los datos de cada alumno.

Desde un punto de vista lógico, este archivo estará formado de un conjunto indeterminado de registros (véase la pág. 54), correspondientes a tantas personas como se desee. La estructura de este registro se define como sigue:



Esta estructura es muy limitada para ser de verdadera utilidad práctica, pues no permite sino un número fijo (y pequeño) de caracteres para cada campo, pero será suficiente para nuestras finalidades, que de antemano restringimos.

El sistema tendrá los siguientes módulos funcionales:

- Verificación de existencia de los archivos.
- Integración de nuevos datos.

- Eliminación de algún dato.
- Alteraciones en un dato ya existente.
- Impresión del archivo.
- Búsqueda por nombre.

que serán controlados por un programa principal. A continuación se describe un primer acercamiento a su diseño.

Verificación de existencia del archivo

Aquí se escribe esta función como módulo separado, aunque en la codificación puede cambiar, o incluso desaparecer, debido a las facilidades de los lenguajes de programación para efectuarla.

```
proc existe( archivo, resultado )
resultado = SI
preguntar al sistema operativo si el archivo ya existe
dentro del sistema de archivos de la computadora.
si no existe
  entonces  comienza
    escribe "ERROR: el archivo", archivo, "aún no ha sido creado."
    resultado = NO
  termina
regresa
fin.
```

Integración de nuevos datos

Este módulo puede, como se decía, hacer uso de las facilidades adicionales para el manejo de archivos de que disponga el sistema operativo. Para nuestros fines, bastará con que añada el nuevo dato al final del archivo principal y grabe una marca especial en el archivo de control, para indicar que se ha roto el ordenamiento alfabético (ya que la nueva entrada no necesariamente quedó en el lugar adecuado, sino simplemente al final).

```
proc altas
! Verificar que existan los archivos maestro y de control
llama existe( maestro, resultado )
si resultado = NO entonces llama crea( maestro )
llama existe( control, resultado )
si resultado = NO entonces llama crea( control )
repite
comienza
  lee el registro a ser añadido
  ! Cuidado con datos duplicados.
  llama busca( nombre, está )
  si está = SI entonces escribe "Error: el nombre ya existe."
              otro grabar el registro en el archivo maestro
termina
hasta( no se deseen añadir más datos )
! "Avisa" que el archivo no está ordenado.
graba una marca en el archivo de control
regresa
fin.
```

Eliminación de algún dato

Esta es la función inversa del módulo de integración, y también puede llegar a ser bastante compleja, dependiendo de la cantidad de recursos del sistema operativo que se empleen en su diseño. Aquí se usará una función llamada borra, que se explica más adelante.

```
proc bajas
! Verificar que exista el archivo maestro.
llama existe( maestro, resultado )
si resultado = NO entonces regresa
repite
comienza
  lee el registro a ser eliminado
  llama busca( nombre, está )
  si está = SI entonces "borrar" ese registro del archivo
              otro escribe "Error: nombre no registrado."
termina
hasta( no se deseen eliminar más datos )
regresa
fin.
```

Alteración de un dato ya existente

En este módulo se usa la capacidad que tienen prácticamente todos los sistemas de manejo de archivos para leer o escribir un registro en particular dentro de un archivo, y que más adelante se explica con detalle.

```
proc cambios
! Verificar que exista el archivo maestro.
llama existe( maestro, resultado )
si resultado = NO entonces regresa
repite
comienza
  lee la modificación deseada en el registro
  llama busca( nombre, está )
  si está = SI entonces modificar ese registro del archivo
              otro escribe "Error: nombre no registrado."
termina
hasta( no se deseen cambiar más datos )
regresa
fin.
```

Impresión del archivo

El diseño de este módulo es muy sencillo, pero falta considerar que la impresión esté ordenada (alfabéticamente, por ejemplo), lo cual puede ser complejo.

En una versión más complicada del sistema podría suponerse que se tiene acceso a una rutina del sistema operativo que realiza precisamente esa función: recibir como entrada un archivo y producir como salida el mismo archivo, ya ordenado. Para esto se definiría también un pequeño archivo de control, que contendrá una marca que indique si el archivo maestro está ordenado o no. El archivo maestro requiere un nuevo ordenamiento si se ha añadido una nueva entrada, quedando como tarea del módulo de integración de nuevos datos avisar (por medio de la marca) si se ha alterado el ordenamiento original.

```
proc impresión
! Módulo de impresión del archivo completo.
! Verificar que exista el archivo.
llama existe( maestro, resultado )
si resultado = NO entonces regresa
! Determinar si está o no ordenado.
si el archivo de control así lo indica entonces
comienza
  llama ordena( maestro )
  grabar la marca de "ordenado" en el archivo de control
termina
mientras (haya registros)
comienza
  lee un nuevo registro del archivo maestro
  tomar nota de la calificación, para calcular el promedio
  imprime registro
termina
regresa
fin.
```

Como puede apreciarse, el sistema no es complejo, pero este diseño depende en buena parte de los recursos que ofrezcan el sistema operativo y el lenguaje de programación para efectuar funciones como buscar, borrar y leer un registro, y otras similares.

Lo importante aquí es la capacidad de localizar un registro dentro de un archivo guiados únicamente por el contenido de cierto campo. En términos generales, un campo que sirve para llegar hasta un registro se conoce como **llave**. En principio, cualquier campo puede funcionar como llave en un registro. Dependiendo de lo complejo que sea un sistema habrá más o menos llaves de acceso, que permitirán una mayor o menor flexibilidad en su uso. Para este caso, tan sólo el campo NOMBRE servirá como llave y, por tanto, no se podrán realizar búsquedas guiadas por el número telefónico, por ejemplo.

El manejo completo de posibilidades de acceso a archivos de datos es muy amplio, y queda fuera del alcance de este libro, por lo que aquí nos limitaremos a trabajar de manera sencilla con el sistema propuesto, explicando al mismo tiempo los conceptos más importantes en el manejo de archivos. (En el apartado 4.7 se mencionaron los sistemas manejadores de bases de datos, que llevan estos conceptos de manejo de archivos a su expresión más amplia y elaborada.)

Existen básicamente dos tipos de archivos: **secuenciales** y **de acceso directo**. En los primeros, la información existente se recupera en el estricto orden en que fue creada, y la nueva se añade siempre al final, mientras que en los segundos existe un número de subesquemas para lograr una mayor flexibilidad que permita añadir o recuperar información sin tener que ceñirse a un orden prefijado.

Intuitivamente, los archivos secuenciales son más sencillos de usar que los otros, porque se espera menos de ellos. De hecho, los métodos de manejo de archivos no secuenciales son muy variados y complejos (*Index Sequential* (ISAM), *Hashed, Keyed Index Sequential* (KSAM) , Árboles binarios, *B* Trees*, etc.), razón por la cual no se trabajará con ellos aquí.

A continuación se describen los pasos necesarios para efectuar algunas operaciones con archivos secuenciales.

Operaciones sobre archivos secuenciales

LOCALIZAR UN REGISTRO, DADA UNA LLAVE DE ACCESO:
Recorrer todo el archivo en forma secuencial hasta encontrar el primer registro cuya llave concuerde con la pedida.

AÑADIR UN REGISTRO:
Localizar el fin del archivo y escribir a partir de ahí el registro en cuestión.

ELIMINAR UN REGISTRO, DADA UNA LLAVE DE ACCESO:
No hay manera eficiente de hacer esto. Lo único que se puede hacer es simplemente marcar el registro como "no usado".

Esto último es grave, y constituye una buena razón para el estudio de métodos más complejos de manejo de archivos. Cuando se opta por mar-

car un registro como inexistente, éste sigue ocupando espacio, que debe considerarse como desperdiciado e inutilizable. Está claro que puede darse el caso de que un archivo consista mayoritariamente en registros inutilizables, lo cual es costoso e ineficiente. El siguiente esquema de **compactación** debe emplearse para eliminar (físicamente) estos "huecos" y volver el archivo a su condición inicial:

```
proc compactación
! Se supone la existencia del archivo original, que se desea compactar,
! y de un archivo temporal.
! Al final del proceso, el archivo original se elimina y el archivo
! temporal se convierte en el nuevo original, ya compactado,
posiciónate al inicio del archivo original
repite
comienza
  lee un nuevo registro del archivo original
  si no está marcado entonces grábalo en el archivo temporal
termina
hasta (que no haya más registros en el archivo original)
elimina el archivo original
"renombra" el archivo temporal con el nombre del original
fin.
```

También resulta claro que el procedimiento de compactación es costoso, pues se tiene que leer todo el archivo original, que puede ser muy grande; no obstante, puede decirse lo mismo del procedimiento de búsqueda de un registro determinado.

A pesar de todas sus desventajas, un sistema secuencial de archivos puede utilizarse con relativa economía cuando es de dimensiones pequeñas, como en este ejemplo, y cuando no se espera darle un uso demasiado "conversacional", sino más bien orientado hacia la generación de informes globales.

Si estas limitantes no se cumplen, entonces se vuelve necesario usar algoritmos más complejos, como ya se mencionó. Sin embargo, no hay que tener la impresión de que los métodos más complejos son baratos, o que consumen pocos recursos computacionales. De hecho, algunos de los esquemas más poderosos y flexibles para el manejo de grandes bancos de información son extremadamente ávidos de espacio en disco.

Sin demasiada complejidad, existe por otro lado la posibilidad de llegar directamente a un registro, no guiados por una llave de acceso (cosa que vuelve complejo al sistema), sino simplemente **por su posición relativa** dentro del archivo completo. Para esto es necesario, claro, que todos los registros sean de longitud fija y conocida. Tanto FORTRAN como Pascal permiten decir, por ejemplo, "leer el registro número 27 del archivo", sin necesidad de pasar secuencialmente por los primeros 26. Basados en esta idea, existe un número de algoritmos para el manejo de archivos que representan un compromiso entre la sencillez y la relativa torpeza de los métodos secuenciales, por un lado, y la sofisticación y complejidad de los esquemas más flexibles, por el otro. Tal es el esquema que se adopta en el módulo de cambios.

Si se usara con un sistema administrador de bases de datos, como se ha dicho, se eliminaría por completo la necesidad de trabajar (desde el punto de vista del usuario) con archivos y métodos de acceso, reemplazándolos por consideraciones más bien de tipo semántico; más cercanas al problema mismo y no a sus detalles técnicos.

Pero sigamos con el sistema. Este es el programa principal, que maneja y manda llamar a los módulos ya expuestos:

```
programa ALUMNOS
repite
comienza
  escribe "PEQUEñO SISTEMA DE REPORTES"
  escribe " Estas son las operaciones disponibles:"
  escribe " A. ALTAS "
  escribe " B. BAJAS "
  escribe " C. CAMBIOS "
  escribe " D. IMPRESION DEL ARCHIVO "
  escribe " F. FIN "
  escribe " Digite su opción: _ "
  lee opción
  caso opción de
    "A" : llama altas
    "B" : llama bajas
    "C" : llama cambios
    "D" : llama impresión
    "F" : ;
        : escribe "Opción desconocida"
  fin-caso
termina
hasta( opción =  "F" )
fin.
```

Antes de proceder a la codificación de cada módulo, advertiremos al lector que los diversos compiladores y sistemas operativos disponen de operaciones e instrucciones que no siempre están estandarizadas, por lo que es casi seguro que los detalles cambien de una instalación a otra. En términos generales, para poder utilizar un archivo ya existente es necesario "abrirlo" por medio de alguna llamada al sistema operativo, así como "cerrarlo" cuando se ha terminado de usarlo.

Nuestro FORTRAN dispone de las siguientes instrucciones para el manejo de archivos, que casi siempre existen en las demás versiones:

```
OPEN(  UNIT = nn,   FILE = <nombre>,   STATUS = <opción>,
RECL = mm,   ERR = eeeee )
```

que sirve para abrir un archivo y leer o escribir en él.

- UNIT especifica el dispositivo físico de almacenamiento *nn* que será usado por las instrucciones READ y WRITE .

- FILE permite especificar el nombre del archivo.
- STATUS tiene las siguientes <opciones>:
  — O *ld*: especifica que el programa apunta al inicio del archivo, que debe existir. Si no es así, hay error.
  — N *ew*: especifica que el archivo no existe, y que hay que abrirlo y colocarse al inicio.
  — A *ppend*: especifica que el archivo debe existir, y que hay que posicionarse al final.
- RECL especifica que los registros son de longitud fija, de tamaño *mm*.
- ERR permite una transferencia a la etiqueta *eeeee* si hubo algún error.

Además, existen las siguientes dos:

```
READ( < UNIT > , < FMT > , REC = mm, END = eeeee )
```

y su opuesta

```
WRITE( < UNIT > , < FMT > , REC = mm, END = eeeee )
```

- UNIT es la especificada por la instrucción OPEN.
- FMT es la etiqueta de un formato de lectura o escritura.
- REC pide leer o escribir el registro *mm* -ésimo del archivo, de forma directa (es opcional pero, si se usa, OPEN tuvo que haber usado la opción RECL).
- END permite una transferencia a la etiqueta *eeeee* si se llegó al fin del archivo (es opcional).

Sin emplear las opciones de ordenamiento, ésta es la codificación del sistema en FORTRAN:

```
C     PROGRAMA "alumno.f"
      INTEGER OPCION
      WRITE(*,*)  ""
C                        ESTRUCTURA TIPO 'CASO'
100   CONTINUE
      WRITE(*,*)
      WRITE(*,*) "              PEQUEÑO SISTEMA DE REPORTES"
      WRITE(*,*)
      WRITE(*,*) "     ESTAS SON LAS OPCIONES DISPONIBLES:"
      WRITE(*,*)
      WRITE(*,*) "     A. ALTAS"
      WRITE(*,*) "     B. BAJAS"
      WRITE(*,*) "     C. CAMBIOS"
      WRITE(*,*) "     D. IMPRESION DEL ARCHIVO"
      WRITE(*,*) "     F. FIN"
110   WRITE(*,*)
      WRITE(*,*) "     DIGITE SU OPCION:"
      READ(*,20) OPCION
      IF( OPCION .NE. 'A' ) GOTO 120
      CALL ALTAS
      GOTO 100
120   IF( OPCION .NE. 'B' ) GOTO 130
      CALL BAJAS
      GOTO 100
130   IF( OPCION .NE. 'C' ) GOTO 140
      CALL CAMBIO
      GOTO 100
140   IF( OPCION .NE. 'D' ) GOTO 150
      CALL IMPRE
      GOTO 100
150   IF( OPCION .EQ. 'F' ) GOTO 200
C                  ERROR EN LA OPCION
      GOTO 110
200   CONTINUE
 20   FORMAT(1A1)
      END
```

La única observación en el programa principal, además de que se están usando las simulaciones de las estructuras de control anteriormente descritas, son los renglones

`READ(*,20) OPCION`, que pide leer el valor de la variable `OPCION` y
`20 FORMAT(1A1)`, que especifica un formato de la lectura alfanumérico.

Como se dijo anteriormente, cuando se desea leer/escribir algún dato mediante la terminal, se usan las instrucciones `READ(*,*)` o `WRITE(*,*)`, sin necesidad de especificar un formato en particular.

Sugerimos al lector que consulte el manual de FORTRAN para estudiar estos detalles, que no seguiremos especificando aquí, por considerarlos poco trascendentes para nuestros fines, aunque deben de estar totalmente especificados para que el programa pueda ser compilado y ejecutado.

```
      SUBROUTINE ALTAS
C     INTEGER CONTROL
      INTEGER ALUMNO(30), CALIF, REG
      LOGICAL ESTA, RESULT
      WRITE(*,*)  ""
      WRITE(*,*)
      WRITE(*,*) " MODULO DE ALTAS"
      WRITE(*,*)
      OPEN( UNIT=7, FILE='MAESTRO', STATUS='O', RECL=33, ERR=180 )
C                 ITERACION TIPO -REPITE-
100   CONTINUE
      WRITE(*,*)
      WRITE(*,*)
      WRITE(*,*) " DIGITE EL NOMBRE DEL NUEVO ALUMNO (30 LETRAS MAX.)"
      WRITE(*,*) " PARA TERMINAR, DIGITE 0"
      READ(*,30) ( ALUMNO(I), I = 1,30 )
      IF( ALUMNO(1) .EQ. '0' ) GOTO 200
110   CONTINUE
      WRITE(*,*) " DIGITE SU CALIFICACION (0-100):"
      READ(*,*) CALIF
      IF( CALIF .LT. 0 .OR. CALIF .GT. 100 ) GOTO 110
      CALL BUSCA( ALUMNO, ESTA, REG )
      IF ( ESTA ) GOTO 120
 C                  EL NUEVO NOMBRE SE ESCRIBE AL FINAL DEL ARCHIVO
        OPEN( UNIT=7, FILE='MAESTRO', STATUS='A', RECL=33, ERR=190 )
        WRITE(7,40) ( ALUMNO(I), I = 1,30 ), CALIF
      GOTO 100
120   WRITE(*,*) "ERROR: EL NOMBRE YA EXISTE."
      GOTO 100
C
      WRITE(*,*)
180   WRITE(*,*) "  EL ARCHIVO NO EXISTIA; SE CREA EN ESTE MOMENTO."
      WRITE(*,*)
      OPEN( UNIT=7, FILE='MAESTRO', STATUS='N', RECL=33, ERR=190 )
      GOTO 100
190   WRITE(*,*) " ALTAS: ERROR EN EL ARCHIVO."
      WRITE(*,*)
      RETURN
200   CONTINUE
 30   FORMAT(30A1)
 40   FORMAT(30A1,I3)
      END
```

Hay un renglón que cumple las funciones del módulo que determina la existencia del archivo:

```
OPEN( UNIT = 7,  FILE = 'MAESTRO',  STATUS = 'O',  RECL = 33,  ERR = 180 )
```

que intenta abrir el archivo MAESTRO, y ejecuta lo que diga el renglón con etiqueta 180 si es que aún no ha sido creado; allí se vuelve a abrir el archivo, ahora con la opción *New*, que lo crea en ese momento.

Si el archivo ya existía, entonces se ejecuta otro OPEN, con la opción Append, que permite añadir nuevos registros al final del archivo.

Otra observación importante es la manera un poco primitiva en que FORTRAN maneja cadenas de caracteres: mediante un vector entero, en donde cada carácter ocupa un espacio del arreglo. Así, es necesario leer y escribir carácter por carácter (con formato Alfanumérico), controlando las operaciones por medio de un DO implícito, como en el renglón

```
READ(*,30) ( ALUMNO(I), I = 1,30 )
```

que lee hasta treinta caracteres alfanuméricos (dígitos o letras), y los coloca en el arreglo ALUMNO, uno por uno.

Por último, la variable ESTA fue declarada como LOGICAL, lo cual significa que es una variable booleana, que solamente puede tomar dos valores: .TRUE. o .FALSE.

```
      SUBROUTINE BUSCA( ALUMNO, ESTA, REG )
      INTEGER ALUMNO(30), CALIF, DATO(30), REG
      LOGICAL ESTA
      REG = 1
100   CONTINUE
      READ(7,40,REC=REG,END=200) ( DATO(I), I = 1,30 ), CALIF
C                  SE COMPARAN LOS NOMBRES.
      DO 110 I = 1, 30
         IF( ALUMNO(I) .NE. DATO(I) ) GOTO 150
110   CONTINUE
C                  LO ENCONTRE.
      ESTA = .TRUE.
      RETURN
C                  SIGO BUSCANDO...
150   CONTINUE
      REG = REG + 1
      GOTO 100
C                  NO ESTUVO.
200   CONTINUE
      REG = 0
      ESTA = .FALSE.
      RETURN
 40   FORMAT(30A1,I3)
      END
```

Esta subrutina de apoyo recibe una cadena de caracteres como parámetro (ALUMNO), y la compara con todos los registros del archivo (identificado por medio de la unidad 7 del READ), usando una variable auxiliar llamada DATO.

Estudie el lector con cuidado el algoritmo, que emplea un DO para comparar carácter por carácter el registro pasado como parámetro con el registro recién leído del archivo. Cuando se detecta la primera desigualdad se abandona el DO y se lee un nuevo registro. Al final, o se encontró una pareja (y la variable ESTA se vuelve .TRUE.) o bien se agotó el archivo y no se encontró lo buscado (y la variable ESTA se vuelve .FALSE. para indicar el resultado final de la subrutina).

La variable REG apunta al registro recién leído, y servirá como parámetro que indica en qué posición dentro del archivo se encontró el registro buscado. Si la búsqueda fracasa, entonces se hace igual a cero, por convención.

```
      SUBROUTINE BAJAS
      INTEGER ALUMNO(30), REG
      LOGICAL ESTA
C                    CARACTER ESPECIAL DE "BORRADO"
      DATA MARCA / '*' /
      WRITE(*,*)  ""
      WRITE(*,*)
      WRITE(*,*) " MODULO DE BAJAS"
      WRITE(*,*)
      OPEN(UNIT=7, FILE='MAESTRO', STATUS='O', RECL=33, ERR=300)
      NUM = 0
C                    ITERACION TIPO -REPITE-
100   CONTINUE
      WRITE(*,*)
      WRITE(*,*) " DIGITE EL NOMBRE DEL ALUMNO A SER BORRADO."
      WRITE(*,*) " PARA TERMINAR, DIGITE 0"
      READ(*,30) ( ALUMNO(I), I = 1,30 )
      IF( ALUMNO(1) .EQ. '0' ) GOTO 200
      CALL BUSCA( ALUMNO, ESTA, REG )
C                    "MARCA" EL REGISTRO COMO "INEXISTENTE"
      IF( .NOT. ESTA ) GOTO 150
        WRITE(7,50,REC=REG) MARCA
        WRITE(*,*) " FUE BORRADO."
        NUM = NUM + 1
        GOTO 100
150   WRITE(*,*) " ERROR: NOMBRE NO REGISTRADO."
      GOTO 100
200   CONTINUE
      WRITE(*,*) " SE BORRARON ", NUM, " NOMBRE(S) DEL ARCHIVO."
      RETURN
C
300   CONTINUE
      WRITE(*,*) " ERROR: EL ARCHIVO AUN NO EXISTE."
      WRITE(*,*)
      RETURN
 30   FORMAT(30A1)
 50   FORMAT(1A1)
      RETURN
      END
```

Esta rutina emplea acceso directo al archivo, porque pone una marca en el registro que se desea borrar. Para lograr esto hay que ir directamente al registro en cuestión, por medio del parámetro `REG` que devolvió la rutina de búsqueda.

La marca es un carácter especial, que no debe confundirse con ninguna de las letras que forman un nombre cualquiera. (En una aplicación real de un sistema así habría que asegurar —no sólo esperar o suponer— que la marca no puede ser incluida en un nombre por equivocación.)

```
      SUBROUTINE CAMBIO
      INTEGER ALUMNO(30), CALIF, REG
      LOGICAL ESTA
      WRITE(*,*)  ""
      WRITE(*,*)
      WRITE(*,*) " MODULO DE CAMBIOS"
      WRITE(*,*)
      OPEN(UNIT=7, FILE='MAESTRO', STATUS='O', RECL=33, ERR=300)
      NUM = 0
C                  ITERACION TIPO -REPITE-
100   CONTINUE
      WRITE(*,*) " DIGITE EL NOMBRE DEL ALUMNO CUYA CALIFICACION"
      WRITE(*,*) " SE DESEA ALTERAR."
      WRITE(*,*) " PARA TERMINAR, DIGITE 0"
      READ(5,30) ( ALUMNO(I), I = 1,30 )
      IF( ALUMNO(1) .EQ. 'O' ) GOTO 200
      CALL BUSCA( ALUMNO, ESTA, REG )
      IF( .NOT. ESTA ) GOTO 150
        READ(7,40,REC=REG) ( ALUMNO(I), I = 1,30 ), CALIF
        WRITE(*,*) " SU CALIFICACION ES ", CALIF
110     CONTINUE
        WRITE(*,*) " DIGITE LA CORRECCION (0-100):"
        READ(*,*) CALIF
        IF( CALIF .LT. 0 .OR. CALIF .GT. 100 ) GOTO 110
        WRITE(7,40,REC=REG) ( ALUMNO(I), I = 1,30 ), CALIF
        NUM = NUM + 1
        GOTO 100
150   WRITE(*,*) " ERROR: NOMBRE NO REGISTRADO."
      WRITE(*,*)
      GOTO 100
200   CONTINUE
      WRITE(*,*)." SE ALTERARON ", NUM, " CALIFICACION(ES) DEL ARCHIVO."
      WRITE(*,*)
      RETURN
C
300   CONTINUE
      WRITE(*,*) " ERROR: EL ARCHIVO AUN NO EXISTE."
      RETURN
 30   FORMAT(30A1)
 40   FORMAT(30A1,I3)
      RETURN
      END
```

En realidad, aquí no hay nada nuevo en lo que se refiere al manejo de FORTRAN. Otra vez se emplea lectura y escritura de acceso directo, guiadas por el apuntador REG que devolvió el subprograma de búsqueda.

Este es el código del módulo que lee e imprime el archivo de datos ya creado.

```
      SUBROUTINE IMPRE
      INTEGER ALUMNO(30), CALIF
      REAL PROM, SUMA
C                  CARACTER ESPECIAL DE "BORRADO"
      DATA MARCA / '*' /
      WRITE(*,*)  ""
      SUMA = 0
      WRITE(*,*)
      WRITE(*,*) " MODULO DE IMPRESION DEL ARCHIVO"
      OPEN(UNIT=7, FILE='MAESTRO', STATUS='O', RECL=33, ERR=300)
      WRITE(*,*) " FAVOR DE ESPERAR UN MOMENTO"
      WRITE(*,*)
      NUM = 0
100   CONTINUE
C                  LECTURA SECUENCIAL DEL ARCHIVO
      READ(7,40,END=200) ( ALUMNO(I), I = 1,30 ), CALIF
C                  NO IMPRIME LOS REGISTROS "BORRADOS"
      IF( ALUMNO(1) .EQ.  MARCA ) GOTO 100
C                  IMPRIME 40 LINEAS POR HOJA
      IF( MOD(NUM,40) .NE. 0 ) GOTO 150
        WRITE(*,*)  ""
        WRITE(6,*)
        WRITE(6,*) "         REPORTE DEL ARCHIVO DE CALIFICACIONES"
        WRITE(6,*)
        WRITE(6,*) "     NOMBRE                          CALIFICACION"
        WRITE(6,*)
150   CONTINUE
        NUM = NUM + 1
        WRITE(6,50) NUM, ( ALUMNO(I), I = 1,30 ), CALIF
        SUMA = SUMA + CALIF
      GOTO 100
C
200   CONTINUE
      IF( NUM .NE. 0 ) PROM = SUMA / NUM
      WRITE(6,*)
      WRITE(6,*)
      WRITE(6,*) " EL PROMEDIO TOTAL ES ", PROM
      RETURN
C
300   CONTINUE
      WRITE(*,*) " ERROR: EL ARCHIVO AUN NO EXISTE."
      WRITE(*,*)
      RETURN
 40   FORMAT(30A1,I3)
 50   FORMAT(I3,3X,30A1,5X,I3)
      RETURN
      END
```

En esta rutina se leyó (en forma secuencial) todo el archivo, sin tomar en cuenta aquellos registros que fueron borrados (marcados), para efectos de calcular el promedio pedido.

Este es un ejemplo de los resultados del sistema; aparecen subrayadas las respuestas del usuario (no se muestran todos los nombres que se dieron, para no alargar el ejemplo):

```
                PEQUEÑO SISTEMA DE REPORTES

    ESTAS SON LAS OPCIONES DISPONIBLES:

    A. ALTAS
    B. BAJAS
    C. CAMBIOS
    D. IMPRESION DEL ARCHIVO
    F. FIN

    DIGITE SU OPCION: D
MODULO DE IMPRESION DEL ARCHIVO
ERROR: EL ARCHIVO AUN NO EXISTE.


                PEQUEÑO SISTEMA DE REPORTES

    ESTAS SON LAS OPCIONES DISPONIBLES:

    A. ALTAS
    B. BAJAS
    C. CAMBIOS
    D. IMPRESION DEL ARCHIVO
    F. FIN

    DIGITE SU OPCION: A
MODULO DE ALTAS

 EL ARCHIVO NO EXISTIA; SE CREA EN ESTE MOMENTO.


DIGITE EL NOMBRE DEL NUEVO ALUMNO (30 LETRAS MAX.)
PARA TERMINAR, DIGITE 0
ALFREDO SANCHEZ A.
DIGITE SU CALIFICACION (0-100): 87

        .
        .
        .
```

```
        .
        .
        .

DIGITE EL NOMBRE DEL NUEVO ALUMNO (30 LETRAS MAX.)
PARA TERMINAR, DIGITE 0
ALFREDO SANCHEZ A.
DIGITE SU CALIFICACION (0-100): 87
ERROR: EL NOMBRE YA EXISTE.

        .
        .
        .

DIGITE EL NOMBRE DEL NUEVO ALUMNO (30 LETRAS MAX.)
PARA TERMINAR, DIGITE 0
JOSE QUIROGA
DIGITE SU CALIFICACION (0-100): 88

DIGITE EL NOMBRE DEL NUEVO ALUMNO (30 LETRAS MAX.)
PARA TERMINAR, DIGITE 0
0


            PEQUEÑO SISTEMA DE REPORTES

    ESTAS SON LAS OPCIONES DISPONIBLES:

    A. ALTAS
    B. BAJAS
    C. CAMBIOS
    D. IMPRESION DEL ARCHIVO
    F. FIN

    DIGITE SU OPCION: D


MODULO DE IMPRESION DEL ARCHIVO
FAVOR DE ESPERAR UN MOMENTO
```

```
                REPORTE DEL ARCHIVO DE CALIFICACIONES

     NOMBRE                          CALIFICACION

1    ALFREDO SANCHEZ A.                    87
2    GUILLERMO ARAGONESES                  80
3    GERARDO GONZALEZ                      82
4    ALEJANDRO GONZALEZ H.                 89
5    GUADALUPE RODRIGUEZ                   80
6    PATRICIA VAN RANKIN                   81
7    ALFONSO ZAMARRIPA                     90
8    ARIEL GUZIK G.                        91
9    JOSE QUIROGA                          88


EL PROMEDIO TOTAL ES   85.3333


                PEQUEÑO SISTEMA DE REPORTES

     ESTAS SON LAS OPCIONES DISPONIBLES:

     A. ALTAS
     B. BAJAS
     C. CAMBIOS
     D. IMPRESION DEL ARCHIVO
     F. FIN

     DIGITE SU OPCION: C


MODULO DE CAMBIOS

DIGITE EL NOMBRE DEL ALUMNO CUYA CALIFICACION
SE DESEA ALTERAR.
PARA TERMINAR, DIGITE 0
MARCO ADAMO

ERROR: NOMBRE NO REGISTRADO.

DIGITE EL NOMBRE DEL ALUMNO CUYA CALIFICACION
SE DESEA ALTERAR.
PARA TERMINAR, DIGITE 0
ARIEL GUZIK G.
SU CALIFICACION ES          91
DIGITE LA CORRECCION (0-100): 92
```

```
DIGITE EL NOMBRE DEL ALUMNO CUYA CALIFICACION
SE DESEA ALTERAR.
PARA TERMINAR, DIGITE 0
0

SE ALTERARON          1  CALIFICACION(ES) DEL ARCHIVO.


            PEQUEÑO SISTEMA DE REPORTES

    ESTAS SON LAS OPCIONES DISPONIBLES:

    A. ALTAS
    B. BAJAS
    C. CAMBIOS
    D. IMPRESION DEL ARCHIVO
    F. FIN

    DIGITE SU OPCION: B
MODULO DE BAJAS


DIGITE EL NOMBRE DEL ALUMNO A BORRAR.
PARA TERMINAR, DIGITE 0
GUADALUPE RODRIGUEZ
FUE BORRADO.

DIGITE EL NOMBRE DEL ALUMNO A BORRAR.
PARA TERMINAR, DIGITE 0
0

SE BORRARON           1  NOMBRE(S) DEL ARCHIVO.


            PEQUEÑO SISTEMA DE REPORTES

    ESTAS SON LAS OPCIONES DISPONIBLES:

    A. ALTAS
    B. BAJAS
    C. CAMBIOS
    D. IMPRESION DEL ARCHIVO
    F. FIN

    DIGITE SU OPCION: D
```

MODULO DE IMPRESION DEL ARCHIVO
FAVOR DE ESPERAR UN MOMENTO

REPORTE DEL ARCHIVO DE CALIFICACIONES

	NOMBRE	CALIFICACION
1	ALFREDO SANCHEZ A.	87
2	GUILLERMO ARAGONESES	80
3	GERARDO GONZALEZ	82
4	ALEJANDRO GONZALEZ H.	89
5	PATRICIA VAN RANKIN	81
6	ALFONSO ZAMARRIPA	90
7	ARIEL GUZIK G.	92
8	JOSE QUIROGA	88

EL PROMEDIO TOTAL ES 86.1250

PEQUEÑO SISTEMA DE REPORTES

ESTAS SON LAS OPCIONES DISPONIBLES:

A. ALTAS
B. BAJAS
C. CAMBIOS
D. IMPRESION DEL ARCHIVO
F. FIN

DIGITE SU OPCION: F

También sin emplear las opciones de ordenamiento, esta es la codificación del sistema en Pascal, con la que se obtienen resultados idénticos:

```
{$I-}              (* Evita errores de E/S a tiempo de ejecucion, *)
                   (* permitiendo que los "atrape" el programa.   *)
{include <stdio.h>}
(* Codificacion en Pascal del sistema de reportes *)
program alumnos ;
const AUSENTE  = 10 ;
      BIEN     =  0 ;
      MARCA    = '*****' ;
      NO       =  0 ;
      SI       =  1 ;
      TITULO   =  'MAESTRO' ;
type  alumno   = record            (* estructura del registro *)
                   nombre : string[30] ;
                   calif  : integer
                 end ;
var   disco    : file of alumno ;
      opcion   : char ;
  function busca( var reg : integer ; datos : alumno ) : boolean ;
  var ya : boolean ;
  begin
    reg := 0 ;
    ya := false ;
    reset( disco ) ;
    while ( not eof(disco) ) and ( ya = false ) do
    begin
      seek( disco, reg ) ;
      get( disco ) ;
      if disco^.nombre = datos.nombre then ya := true
                                      else reg := reg + 1
    end ;
    busca := ya
  end ;  (* busca *)
  procedure existe( var result : integer ) ;
                           (* averigua el 'status' del archivo *)
  begin
    reset( disco, TITULO ) ;
    if ioresult = BIEN
              then result := SI
              else if ioresult = AUSENTE
                     then begin
                            result := NO ;
                            writeln ;
                            writeln( ' El archivo aun no ha sido creado.' ) ;
                            writeln
                          end
```

```
                    else writeln( 'ERROR TIPO ', ioresult )
end ;  (* existe *)
                                        (* *)
procedure altas ;
                              (* Integracion de nuevos datos *)
var reg, result : integer ;
    datos : alumno ;
begin
  writeln( '' ) ;
  writeln( '  MODULO DE ALTAS' ) ;
  existe( result ) ;
  if result = NO then
    begin
                              (* Crea el archivo si aun no existe *)
      writeln(' Se crea en este momento. ' ) ;
      rewrite( disco, TITULO )
    end ;
    repeat
      writeln ;
      writeln ;
      writeln( ' Digite el nombre del nuevo alumno (30 letras max.)' ) ;
      writeln( ' Para terminar, digite 0' ) ;
      readln( datos.nombre ) ;
      if datos.nombre <> '0' then
      begin
        repeat
          write( ' Digite su calificacion (0-100): ' ) ;
          readln( datos.calif )
        until datos.calif in [0..100] ;
        if busca( reg, datos ) then writeln( 'ERROR: el nombre ya existe.' )
                               else begin
                                      disco^ := datos ;
                                      put( disco ) ;
                                      close( disco, lock ) ;
                                      reset( disco, TITULO )
                                    end
      end
    until datos.nombre = '0' ;
    close( disco, lock )
end ;  (* altas *)
procedure bajas ;
                              (* Eliminacion de datos *)
var num, reg, result : integer ;
    datos : alumno ;
begin
  writeln( '' ) ;
  writeln( '  MODULO DE BAJAS' ) ;
  num := 0 ;
  existe( result ) ;
```

```
  if result = NO then exit ( bajas ) ;
  repeat
    writeln ;
    writeln ;
    writeln( ' Digite el nombre del alumno a ser borrado' ) ;
    writeln( ' Para terminar, digite 0' ) ;
    readln( datos.nombre ) ;
    if datos.nombre <> '0' then
      if busca( reg, datos )
         then begin        (* Marca el registro como "inexistente" *)
                seek( disco, reg ) ;
                disco^.nombre := MARCA ;
                put( disco ) ;
                num := num + 1
         end
         else writeln( ' ERROR: nombre no registrado.' )
  until datos.nombre = '0' ;
  writeln ;
  writeln ;
  writeln( ' Se borraron ', num, ' nombre(s) del archivo.' )
end ;  (* bajas *)
procedure cambios ;
                          (* Alteracion en los datos *)
var num, reg, result : integer ;
    datos : alumno ;
begin
  writeln( '' ) ;
  writeln( '  MODULO DE CAMBIOS' ) ;
  num := 0 ;
  existe( result ) ;
  if result = NO then exit ( cambios ) ;
  repeat
    writeln ;
    writeln ;
    writeln( ' Digite el nombre del alumno cuya calificacion' ) ;
    writeln( ' se desea alterar.' ) ;
    writeln( ' Para terminar, digite 0' ) ;
    readln( datos.nombre ) ;
    if datos.nombre <> '0' then
      if busca( reg, datos )
         then begin
                seek( disco, reg ) ;
                get( disco ) ;
                          (* Como el -get- avanza la "ventana", es
                             necesario regresarla al lugar original *)
                seek( disco, reg ) ;
                writeln( ' Su calificacion es: ', disco^.calif ) ;
```

```
                repeat
                  write( ' Digite la correccion (0-100): ' ) ;
                  readln( disco^.calif )
                until disco^.calif in [0..100] ;
                put( disco ) ;
                num := num + 1
            end
            else writeln( ' ERROR: nombre no registrado.' )
    until datos.nombre = '0' ;
    writeln ;
    writeln ;
    writeln( ' Se alteraron ', num, ' calificacion(es) del archivo.' )
end ;  (* cambios *)
procedure imprime ;
                              (* Impresion del archivo *)
var num, reg, result : integer ;
    prom, suma : real ;
    datos : alumno ;
begin
  writeln( '' ) ;
  writeln( ' IMPRESION DEL ARCHIVO.' ) ;
  writeln( ' FAVOR DE ESPERAR UN MOMENTO' ) ;
  existe( result ) ;
  if result = NO then exit ( imprime ) ;
                              (* Lectura secuencial del archivo, sin
                                 imprimir los registros "borrados" *)
  num := 0 ;
  reset( disco ) ;
  while not eof( disco ) do
  begin
      datos := disco^ ;
      get( disco ) ;
      if datos.nombre <> MARCA
      then
      begin
        if ( num mod 40 ) = 0      (* Brinco de pagina *)
        then begin
               writeln ;
               writeln ;
               writeln( '          REPORTE DEL ARCHIVO DE CALIFICACIONES' ) ;
               writeln ;
               writeln( ' .      NOMBRE                          CALIFICACION' ) ;
               writeln
             end ;
        num := num + 1 ;
             writeln( num, '  ', datos.nombre, datos.calif ) ;
             suma := suma + datos.calif
           end
      end ;
```

```
    if num <> 0 then prom := suma / num
                else prom := 0 ;
    writeln ;
    writeln ;
    write( ' El promedio total es: ', prom )
  end;  (* imprime *)

  begin   (* programa principal *)
    repeat
      writeln ;
      writeln ;
      writeln( '       PEQUEÑO SISTEMA DE REPORTES' ) ;
      writeln ;
      writeln( '   ESTAS SON LAS OPCIONES DISPONIBLES:' ) ;
      writeln ;
      writeln( '   A. ALTAS' ) ;
      writeln( '   B. BAJAS' ) ;
      writeln( '   C. CAMBIOS' ) ;
      writeln( '   D. IMPRESION DEL ARCHIVO' ) ;
      writeln( '   F. FIN.' ) ;
      writeln ;
      write( '   DIGITE SU OPCION: ' ) ;
      readln( opcion ) ;
      if opcion in ['A'..'D','F']
        then
          case opcion of
            'A': altas ;
            'B': bajas ;
            'C': cambios ;
            'D': imprime ;
            'F':  ;
          end
        else begin
               writeln ; writeln( ' OPCION DESCONOCIDA. ')
             end
    until opcion = 'F' ;
    close( disco, lock )
  end.
```

Entre las cosas nuevas en esta codificación destacan los renglones

```
type    alumno = record              (* estructura del registro *)
                  nombre : string[30] ;
                  calif  : integer
                end ;
var    disco  : file of alumno;
```

que declaran un archivo (`file`) "de tipo alumno". La declaración del nuevo tipo (`type`) `alumno` es en términos de la palabra reservada `record` de Pascal, que se usa para definir estructuras más complejas que los arreglos, puesto que pueden tener componentes de diversas características; en este caso, cada registro consta de un campo alfanumérico de hasta treinta caracteres, seguido de la calificación, en forma de un número entero.

Una vez declarado este nuevo tipo de variable de Pascal, es posible definir otras nuevas en términos de ella, dentro de la declaración `var` que debe seguir, formando así toda una red de variables complejas (definidas en este esquema de "entrelazamiento"). Esta es una característica importante de Pascal que permite (a diferencia de FORTRAN, por ejemplo) un uso complejo de las estructuras de datos. Este recurso, sin embargo, es un arma de dos filos, ya que si se abusa de él pueden llegarse a escribir programas difíciles de entender, debido precisamente al alto grado de interdependencia de las variables.

En este caso se ha hecho un uso muy sencillo del `type`, y se emplearon variables que dependen de `alumno`, como en el renglón

```
var datos : alumno ;
```

usado para declarar una variable local para leer datos.

Archivos de acceso directo en Pascal

Para manejar archivos en Pascal es necesario declararlos por medio de la palabra `file`, de la forma mostrada. Al hacer esto, el compilador define automáticamente una nueva variable (formada con el nombre del archivo y el símbolo ^ —una pequeña flecha que apunta hacia arriba—), que sirve como "ventana" al archivo, ya que permite observar un registro completo, para fines de lectura o escritura.

Así pues, la variable especial `disco^` actúa como ventana, y es la forma mediante la cual se puede leer o escribir en ese archivo. Las operaciones de lectura/escritura se hacen por medio de las instrucciones `get/put`, que obtienen/depositan el contenido de esta ventana en el archivo.

Como se definió que `disco` es un archivo de registros especiales ("de tipo `alumno` ") y `alumno`, a su vez, está definido con dos componentes de forma diferente entre sí, en ocasiones será necesario expresar con cuál de las dos partes del registro interesa trabajar. Esto se logra mencionando la parte en cuestión antecedida por el nombre del registro, y separada de éste con un punto. De esta forma, `disco^.nombre` es la "subventana" que apunta al primer componente del registro del archivo `disco`.

Usando las variables definidas en el programa, ésta es la secuencia necesaria para leer del archivo y, por ejemplo, imprimir la calificación:

```
get( disco ) ; (* Observese que se usa tan solo el nombre del *)
               (* archivo, sin la flechita al final *)
writeln( disco^.calif ) ; (* Aqui se manda imprimir la subventana *)
                          (* recien extraida del archivo *)
```

La instrucción `get` extrae el contenido de la ventana del archivo que tenga entre paréntesis y lo pone automáticamente en la variable especial terminada

con la flechita, para ser usada luego. Si se está hablando de archivos secuenciales, **get** simplemente obtiene el siguiente registro del archivo. Si ya no hay más, entonces la ventana queda indefinida, y la variable booleana predefinida **eof(disco)** se vuelve verdadera (valor lógico 1). Esta variable, llamada *end of file* (fin de archivo), es controlada en forma interna por la computadora.

Además, **get** siempre adelanta la ventana un registro más dentro del archivo en que se use; es decir, logra dos acciones: primero extrae el registro del archivo (y lo coloca en la variable especial) e inmediatamente después apunta al siguiente.

Para escribir, por ejemplo, se usa la secuencia:

```
disco^.calif := 100 ;
put( disco ) ;
```

es decir, primero se apunta la ventana (o subventana) a los valores deseados, y luego se escribe su contenido en el disco. Por definición, en los archivos secuenciales de Pascal, un **put** solamente puede ocurrir al final del archivo, cuando **eof** es verdadero.

Por otro lado, las intrucciones equivalentes al OPEN de FORTRAN son:

```
reset ( <archivo> , <nombre> ) ;
```

que crea el archivo declarado internamente (**:file of ...**), y que quedará en el disco magnético de la computadora con el nombre que viene luego de la coma. La variable interna **ioresult** devuelve un valor que depende del resultado de la operación y que puede usarse para investigar si el archivo ya existía o no en el sistema de archivos de la computadora. Los detalles específicos dependen del compilador que se use.

```
reset ( <archivo> ) ;
```

que simplemente abre el archivo (que ya debe existir), colocándose al comienzo del mismo.

```
rewrite ( <archivo>, <nombre> ) ;
```

que se usa una sola vez, luego del primer **reset** que crea el archivo, para limpiarlo y poder comenzar a escribir sobre él.

Cuando se ha terminado de usar un archivo (o de escribir un nuevo registro, dependiendo de las instrucciones específicas del compilador que se use), hay que cerrarlo, por medio de la instrucción

```
close ( <archivo>, lock) ;
```

Como se advirtió inicialmente al lector, estos son detalles que no están estandarizados en Pascal, por lo que se vuelve imprescindible consultar el manual de la máquina con la que se trabaje.

Todo esto, que parece complicado, es para archivos secuenciales. Como en el caso de FORTRAN, la mayoría de los compiladores de Pascal permiten el manejo directo de archivos, mediante el cual se puede llegar a un registro en particular sin pasar por todos los anteriores, lo cual se logra con la instrucción

```
seek ( <archivo>, <número> ) ;
```

donde `<número>` es una variable entera que especifica a cuál registro del archivo se va a referir la próxima instrucción `get` o `put`, que necesariamente debe aparecer luego, aunque no sea inmediatamente a continuación.

Por otro lado, si se han definido variables "del tipo de" otras (por medio de `type`), entonces se pueden manejar como un todo, sin tener acceso específico a sus partes internas y como si fueran variables simples. Por ejemplo

```
datos : = disco^ ;
```

copia la información de la ventana del archivo (tal vez recién leída) a la variable `datos`. Ambas son de tipo `alumno`, por lo que lo anterior es simplemente una abreviatura de:

```
datos.nombre := disco^.nombre ;
datos.calif := disco^.calif ;
```

Existe además una instrucción (llamada `with`) que es útil para manejos de estructuras tipo `record`, pero que ya no se verá aquí.

Pedimos al lector que estudie cuidadosamente el programa en Pascal para que, ayudado por el pseudocódigo, avance en la comprensión de estos puntos.

Este sistema, como todos los programas y ejemplos del libro, fue compilado y ejecutado en nuestra computadora, y entrega los mismos resultados que la versión de FORTRAN.

Resta tan sólo terminar este largo capítulo con la misma recomendación que hiciéramos al final del apartado 7.3: practique cuanto pueda.

Recome
final

Palabras y conceptos clave

En esta sección se agrupan las palabras y conceptos de importancia que se estudiaron en el capítulo, con el objetivo de que el lector pueda hacer una autoevaluación que consiste en ser capaz de describir con cierta precisión lo que cada término significa, y no sentirse satisfecho hasta haberlo logrado. Se muestran en el orden en que se describieron o se mencionaron.

CONDICIÓN BOOLEANA
COMMON
VARIABLE LOCAL
VARIABLE GLOBAL
PASO DE PARÁMETROS
POR VALOR

MANEJO DE BLOQUES
ALCANCE DE UNA VARIABLE
PASO DE PARÁMETROS POR REFERENCIA
LLAVE EN UN REGISTRO ISAM

Ejercicios

1. ¿Cuántas soluciones existen para el problema de las ocho damas? Para averiguarlo, modifique el programa de modo que encuentre todas las soluciones (es preferible que no las imprima, sino que sólo indique cuántas son) y ejecútelo en su computadora.
2. Codifique los programas que se pidieron en los ejercicios 1 a 4 del capítulo anterior y ejecútelos en su computadora.
3. Escriba un programa en Pascal (o en FORTRAN) para simular una máquina de Turing, tomando como base el programa en pseudocódigo pedido en el ejercicio 5 del capítulo anterior.
4. En el ejercicio 10 del capítulo anterior se pedía modificar el pseudocódigo del problema de las ocho damas para que encontrara las soluciones que son independientes de la rotación del tablero. Codifique esta nueva versión e imprima las soluciones únicas, varias por página.
5. Escriba un programa en Pascal (o en FORTRAN) para traducir números arábigos a números romanos. El programa debe emplear la representación abreviada para los números 4, 9, 40, 90, etc. Es decir, si recibe como entrada el número 34, por ejemplo, debe producir como resultado XXXIV y no XXXIIII.

   Este es un buen ejemplo de un programa en el que se debe diseñar cuidadosamente la relación entre el algoritmo (escrito, como siempre, inicialmente en pseudocódigo) y las estructuras de datos, ya que una buena elección de éstas producirá un programa sencillo y conciso. Un ejemplo extremo de cómo el algoritmo podría ser casi inexistente y las estructuras de datos complejas sería una gran tabla que contuviera la representación de todos los números menores que 5000, en donde el algoritmo simplemente localizara el número pedido y mostrara su equivalencia. Otro ejemplo, en el extremo contrario, sería un algoritmo complejo que trabajara sólo sobre los caracteres I, V, X, C, M y D, y que los agrupara según se requiriera. Sin embargo, un algoritmo así tendría que considerar los (múltiples) casos especiales que surgen con las abreviaturas. Es preferible, entonces, encontrar un equilibrio entre el algoritmo y las estructuras de datos (cosa que, además, vale para todo programa).

   Escriba el programa e incluya documentación sobre los refinamientos inciales del pseudocódigo y sobre las estructuras de datos empleadas y su uso.

Referencias para el capítulo 8

[DAVG86] Davis, Gordon y Thomas Hoffmann, *FORTRAN 77: Un estilo estructurado y disciplinado*, McGraw-Hill, México, 1986.
Traducción de la segunda edición de un texto introductorio sobre la versión mejorada de este lenguaje. Incluye un capítulo sobre el manejo de la terminal (y no sólo de las tarjetas perforadas, como todavía se encuentra en muchos libros), y otro sobre las diferencias entre varias de las versiones en uso (FORTRAN IV, 77, WATFOR, WATFIV).

[FAIR87] Fairley, Richard, *Ingeniería de software*, McGraw-Hill, México, 1987.
Traducción de un texto de carácter descriptivo sobre los quehaceres de la ingeniería de software. Tiene capítulos sobre planeación y estimación de costos de proyectos de programación, así como consideraciones un tanto más técnicas sobre diseño y verificación de programas y sistemas.

[JENK74] Jensen, Kathleen y Niklaus Wirth, *Pascal User Manual and Report*, Springer-Verlag, Nueva York, 1974.
Referencia original del lenguaje de programación Pascal. Incluye la definición formal del lenguaje en términos de su gramática. El segundo autor, Wirth, es ampliamente conocido por sus estudios sobre programación estructurada y se ha mencionado en los capítulos anteriores.

[JONW82] Jones, William, *Programming Concepts, A Second Course, with Examples in Pascal*, Prentice-Hall, New Jersey, 1982.
Este libro constituye una magnífica fuente de conocimientos tanto del lenguaje Pascal como de temas específicos de estructuras de datos (listas, pilas, apuntadores, etc.). Puede servir tanto de referencia general sobre programación como de manual de uso del lenguaje Pascal..

[KERB81] Kernighan, Brian y P. J. Plauger, *Software Tools in Pascal*, Addison-Wesley, Massachusetts, 1981.
Adaptación para el lenguaje Pascal del excelente *Software Tools* escrito por los mismos autores, que se empleó como referencia en el capítulo 4 ([KERB76]). Todos los algoritmos del libro original están traducidos a Pascal, con la misma metodología y filosofía, lo cual lo hace valioso.

[KOFE86] Koffman, Elliot, *Pascal. Introducción al lenguaje y resolución de problemas con programación estructurada*, Addison Wesley Iberoamericana, México, 1986.

Traducción de un libro bastante completo sobre programación estructurada en Pascal, que incluye temas sobre recursividad, manejo de archivos y estructuras dinámicas de datos. Todos los ejemplos están codificados en español. Este libro tiene una estructura (y subtítulo) similar a uno de FORTRAN, escrito por el mismo autor, [FRIF84].

[SCHW74] Schick, William y Charles Merz, *FORTRAN IV para ingeniería*, McGraw-Hill, México, 1974.
Traducción de un libro muy completo sobre el lenguaje FORTRAN. Aunque el título indica que es para ingenieros, esto sólo se aplica al tipo de problemas y ejercicios de codificación que presenta, y no tanto al enfoque global del texto. Por tratarse de una obra relativamente antigua, presenta algoritmos en términos de diagramas de flujo que no resultan muy claros cuando se comparan con el pseudocódigo.

[SOMI88] Sommerville, Ian, *Ingeniería de Software*, Addison-Wesley Iberoamericana, México, 1988.
Traducción de otro libro sobre la ingeniería de software, que dedica varios capítulos a las consideraciones de tipo psicológico, además de los usuales, en los que se explican criterios de especificación y diseño de programas y sistemas. Se incluye un apéndice sobre la enseñanza de esta disciplina, en el que se propone un esquema para incluir uno de estos cursos en una licenciatura en computación, junto con ejemplos de proyectos para ser desarrollados por los alumnos.

[YOUE79] Yourdon, Edward y Larry Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, New Jersey, 1979.
Complejo y amplio libro que explica la filosofía de la escuela de diseño de Yourdon, que ya se mencionó en el capítulo 7. Se tratan conceptos que van desde los fundamentos del diseño estructurado hasta aplicaciones sobre estructuras recursivas. Los capítulos 6 y 7 de Yourdon se dedican a los temas de cohesión y acoplamiento entre los programas que configuran un sistema. Libro considerado importante dentro del campo de ingeniería de software.

Apéndice A: el sistema operativo Unix

Unix* es un sistema operativo desarrollado en los Laboratorios Bell, en New Jersey, Estados Unidos. En 1969, un grupo de investigadores se dio a la tarea de crear un entorno de programación que facilitara sus labores internas de investigación y desarrollo. Ken Thompson escribió, con el apoyo de Dennis Ritchie y otros investigadores, un sistema operativo de tiempo compartido, pequeño y de propósito general. Esta primera versión fue escrita en el lenguaje ensamblador de una minicomputadora PDP-7 que ya no usaban, y al año siguiente Ritchie la instaló en una máquina más moderna, una PDP-11, y se dedicó también a escribir el compilador para el lenguaje de programación C, que acababan de diseñar. En 1973, Thompson y Ritchie escribieron el núcleo de Unix en el lenguaje C, rompiendo así con la tradición de escribir sistemas operativos en lenguaje ensamblador; con ello se logró que Unix fuera más portátil y fácil de modificar.

Poco después se concedió el permiso para que algunas instituciones no lucrativas tuvieran acceso a Unix en la versión de la PDP-11, que ya era muy popular en universidades e institutos de investigación, y eso marcó el inicio de una rápida difusión del sistema en todo el mundo. En la actualidad Unix se considera un estándar virtual para computadoras multiusuario, y ha sido adoptado por una gran cantidad de máquinas. Existen varias versiones comerciales del sistema: Unix III, Unix V, Unix BSD, Xenix, etc., pero todas tienen mucho en común.

* Unix es una marca registrada de los Laboratorios Bell.

El nombre Unix proviene de un juego de palabras combinado con la filosofía de su diseño. En 1965, los Laboratorios Bell participaron junto con la compañía General Electric en un proyecto de desarrollo de sistemas operativos, integrado al proyecto MAC del Instituto Tecnológico de Massachusetts (MIT). El objetivo era diseñar un gran sistema multiusuario de nombre "Multics", y que no se concluyó. De la experiencia de Multics se aprendieron muchas cosas que hasta la fecha son importantes en programación de sistemas, pero el proyecto no culminó en parte porque se trataba de un diseño muy amplio y complejo. Thompson, Ritchie y otros participantes en el proyecto Multics aprendieron la lección y años después bautizaron a su nuevo sistema con el nombre Unix, que tiene una connotación contraria a la idea de multiplicidad y complejidad.

Unix y la mayoría de los sistemas que se ejecutan en él están escritos en lenguaje C, y han servido como demostración de que un sistema operativo interactivo y poderoso no necesariamente es grande y caro, ya sea en equipo o en cantidad de código: puede utilizarse en minicomputadoras de costo reducido y se requirió menos de dos años-hombre para el desarrollo inicial del sistema principal. En palabras de sus creadores el objetivo es que, desde el punto de vista del usuario, sea simple, elegante y fácil de usar.

Si se recuerda que la función general de un sistema operativo es controlar y dirigir la operación de la computadora, de forma tal que presente una imagen monolítica y virtual (en contraposición con real o electrónica o ingenieril) ante los usuarios del sistema de cómputo, se estará de acuerdo en que el sistema operativo resulta tan importante como las facilidades físicas y electrónicas que proporcione el equipo.

Lo que se espera de un sistema operativo, como se ha dicho en el capítulo 4, es que sea capaz de atender la operación concurrente de múltiples pedidos de atención por parte de procesos que se están ejecutando en la computadora; que sea capaz de mantener toda la operación bajo control sin perder detalle alguno ni permitir que los procesos interfieran entre sí; que logre un óptimo grado de utilización de los recursos físicos de la máquina (procesador, memoria, periféricos) y, por último, que haga todo esto callada y eficientemente.

Como es fácil comprender, son pocos los sistemas operativos que logran todos estos objetivos, que a veces son incluso autocontradictorios (no se puede esperar, por ejemplo, que el sistema sea potente, inteligente, eficiente y pequeño al mismo tiempo).

La razón de la creciente popularidad de Unix reside en la combinación que logra entre facilidad de uso y eficiencia, además de la gran cantidad de ayudas y utilerías para la programación que tiene incluidas. Con Unix es sencillo lograr comunicación y sincronización entre procesos, lo que requiere de programación dedicada y exclusiva en los lenguajes de control de otros sistemas operativos, o bien es virtualmente imposible de lograr en sistemas más limitados.

La filosofía de operación de Unix está basada en el concepto de *herramientas de software*, visión conceptual que pide que las tareas computacionales se construyan paulatinamente (de manera que podría llamarse genética), donde el sistema aporta un conjunto de operaciones primitivas, que el diseñador usa

para armar aplicaciones que, una vez hechas, pasan a formar parte del acervo de operaciones básicas. Es decir, con un pequeño número de funciones elementales se pueden configurar programas y sistemas completos que cumplan una función específica.

Entre las características del sistema operativo Unix están las siguientes:

Características de Unix

- Es un sistema operativo multiusuario, con capacidad de simular multiprocesamiento y procesamiento no interactivo.
- Está escrito en un lenguaje de alto nivel: C.
- Dispone de un lenguaje de control programable, llamado *Shell*.
- Ofrece facilidades para la creación de programas y sistemas, y un ambiente muy propio para las tareas de diseño de software.
- Emplea manejo dinámico de memoria (por intercambio o por paginación).
- Tiene capacidad de interconexión de procesos.
- Permite comunicación entre procesos.
- Emplea un sistema jerárquico de archivos, con facilidades de protección de archivos, cuentas y procesos.
- Usa un manejo consistente de archivos de diversos tipos.
- Tiene facilidades para redireccionamiento de entradas/salidas.
- Incluye más de un centenar de subsistemas, y varios lenguajes de programación.
- Garantiza un alto grado de portabilidad.

El sistema se basa en un núcleo (conocido como *kernel*) que reside permanentemente en la memoria, y que atiende todas las llamadas del sistema, administra el acceso a los archivos y el inicio o suspensión de las tareas de los usuarios.

Unix permite que los programas sean independientes de los dispositivos periféricos; la salida de cada programa o utilería del sistema puede ser dirigida a archivos en disco, impresoras o terminales, y existe también la posibilidad de comunicación entre procesos para crear conjuntos arbitrarios y complejos de procesos concurrentes cooperativos.

La comunicación con la unidad central de procesamiento en el sistema Unix es por medio del programa especializado de control llamado *Shell*. Shell es un lenguaje de control, un intérprete y un lenguaje de programación, y tiene características que lo hacen sumamente flexible para las tareas de un centro de cómputo. Visto como lenguaje de programación, incluye estas características:

Características de Shell

- Ofrece las estructuras de control normales: secuenciación, iteración condicional, selección, y otras más.
- Paso de parámetros.
- Sustitución textual de variables y cadenas.
- Comunicación bidireccional entre órdenes de Shell.

Shell permite modificar en forma dinámica las características con que se ejecutan los programas en Unix: las entradas y salidas pueden ser redireccionadas hacia archivos, procesos y dispositivos; así mismo, es posible interconectar procesos entre sí, y usuarios diferentes pueden "ver" versiones distintas del sistema operativo debido a la capacidad de Shell para configurar diversos ambientes de ejecución. Por ejemplo, se puede hacer que cierto usuario entre en sesión directamente a ejecutar un programa en particular, y sacarlo del sistema automáticamente al terminar de usarlo. A veces resulta conveniente programar la primera versión de un sistema en Shell para probarlo en forma interactiva por medio del intérprete. De la misma forma, es sencillo automatizar tareas que suelen hacerse en forma manual, tales como agrupamiento de órdenes, ejecución seriada de programas, etcétera.

El sistema de archivos

El sistema de archivos de Unix, por su parte, está basado en un modelo arborescente y recursivo, en donde los nodos pueden ser tanto archivos como directorios, y estos últimos pueden contener a su vez directorios o subdirectorios. Debido a esta filosofía, el manejo del sistema es por medio de muy pocas órdenes, que permiten una gran gama de posibilidades. Todo archivo de Unix está controlado por múltiples niveles de protección, que especifican los permisos de acceso al mismo. La diferencia que existe entre un archivo de datos, un programa, un manejador de entrada/salida o una instrucción ejecutable se refleja en estos parámetros, de modo que el sistema operativo adquiere características de coherencia y elegancia que lo distinguen.

La raíz del sistema de archivos (conocida como *root*) se denota con el símbolo /, y de ahí se desprende un conjunto de directorios que contienen todos los archivos del sistema de cómputo. Cada directorio, a su vez, funciona como la subraíz de un nuevo árbol que depende de él, y que también puede estar formado por directorios o subdirectorios y archivos. Un archivo siempre ocupará el nivel más bajo dentro del árbol, porque de un archivo no pueden depender otros; si así fuera, sería un directorio. Es decir, los archivos son hojas del árbol.

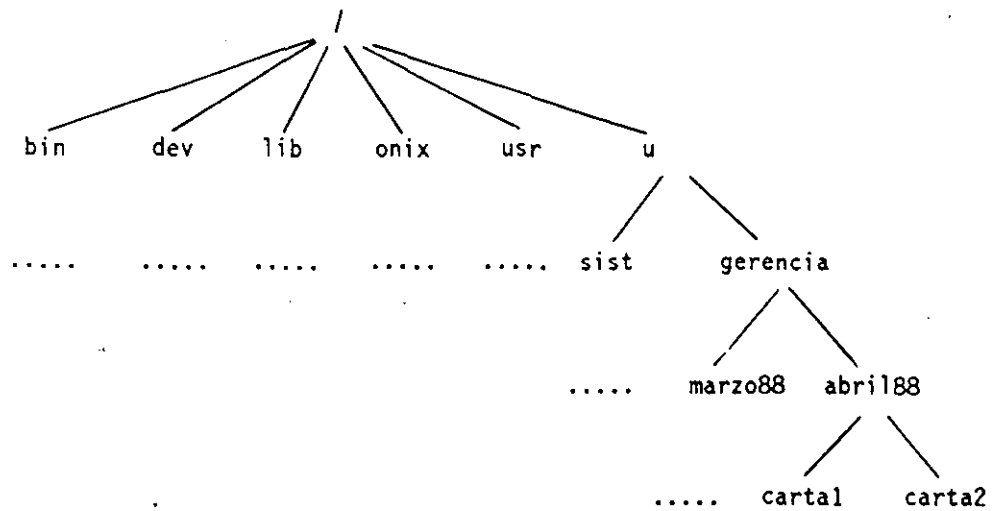
Se define en forma unívoca el nombre de todo archivo (o directorio) mediante lo que se conoce como su **trayectoria** (*path name*): el conjunto completo de directorios, iniciando en *root* (/), por los que hay que pasar para poder llegar al directorio o archivo en cuestión. Cada nombre se separa de los otros con el símbolo /, aunque tan sólo el primero de ellos se refiere a la raíz.

Por ejemplo, el archivo

```
/u/gerencia/abril88/carta2
```

tiene toda esta trayectoria como nombre absoluto, pero se llama `gerencia/abril88/carta2`, sin la diagonal inicial, si se observa desde el directorio `/u`. Para los usuarios que están normalmente en el directorio `/u/gerencia`, el archivo se llama `abril88/carta2`. Así, también puede existir otro archivo llamado `carta2`, pero dentro de algún otro directorio, y en caso de ser necesario se emplearía el nombre de la trayectoria (completa o en partes, de derecha a izquierda) para distinguirlos.

Unix ofrece medios muy sencillos para colocarse en diferentes puntos del árbol que forma el sistema de archivos, que para este caso podría ser el siguiente:



Como se dijo, desde el punto de vista del directorio `abril88`, que a su vez pertenece al directorio `gerencia` del directorio `/u`, basta con el nombre `carta2` para apuntar al archivo en cuestión.

En esta forma se maneja el sistema completo de archivos, y se dispone de un conjunto de órdenes de Shell (además de múltiples variantes) para hacer manipulaciones diversas como crear directorios, moverse dentro del sistema de archivos, copiar archivos, etc. Al final se incluye una descripción de las funciones principales.

Unix incluye además múltiples esquemas para crear, editar y procesar documentos. Existen varios tipos de editores, formadores de textos, macroprocesadores para textos, formadores de tablas, preprocesadores de expresiones matemáticas, y un gran número de ayudas y utilerías diversas, que se mencionan más adelante.

A continuación se describe el modo de funcionamiento de Unix, siguiendo el modelo de estudio de sistemas operativos empleado en el capítulo sobre programación de sistemas.

El núcleo del sistema operativo

El núcleo del sistema operativo Unix (llamado *kernel*) es un programa de aproximadamente 10 000 renglones, escrito casi en su totalidad en lenguaje C, con excepción de una parte del manejo de interrupciones, que está escrita en el lenguaje ensamblador del procesador en el que opera.

Las funciones del núcleo son permitir la existencia de un ambiente en el que sea posible atender a varios usuarios y múltiples tareas en forma concurrente, repartiendo al procesador entre todos ellos, e intentando mantener en grado óptimo la atención individual.

El kernel opera como asignador de recursos para cualquier proceso que necesite hacer uso de las facilidades de cómputo. Es el componente central de Unix y tiene las siguientes funciones:

- Creación de procesos, asignación de tiempos de atención y sincronización.
- Asignación de la atención del procesador a los procesos que lo requieren.
- Administración de espacio en el sistema archivos, que incluye:
  —acceso, protección y administración de usuarios;
  —comunicación entre usuarios y entre procesos, y
  —manipulación de E/S y administración de periféricos.
- Supervisión de la transmisión de datos entre la memoria principal y los dispositivos periféricos.

El kernel reside siempre en la memoria central y tiene el control sobre la computadora, por lo que ningún otro proceso lo puede interrumpir; sólo lo pueden llamar para que proporcione algún servicio de los ya mencionados. Un proceso llama al kernel mediante módulos especiales conocidos como **llamadas al sistema**

El kernel consiste en dos partes principales: la sección de control de procesos y la sección de control de dispositivos. La primera asigna recursos, programa procesos y atiende sus requerimientos de servicio, y la segunda supervisa la transferencia de datos entre la memoria principal y los dispositivos periféricos. En términos generales, cada vez que algún usuario oprime una tecla de una terminal, o cada vez que se debe leer o escribir algo del disco magnético, se interrumpe al procesador central y el núcleo se encarga de efectuar la operación de transferencia.

Cuando se inicia la operación de la computadora se debe cargar en la memoria una copia del núcleo, que reside en el disco magnético. (Se recordará que esta operación inicial recibe el nombre de *bootstrap*.) Para ello, se deben inicializar unas interfaces básicas de hardware, que incluyen el reloj que proporciona interrupciones periódicas. El kernel también prepara algunas estructuras de datos, que incluyen una sección de almacenamiento temporal para transferencia de información entre terminales y procesos, una sección para almacenamiento de descriptores de archivos y una variable que indica la cantidad de memoria principal.

A continuación, el kernel inicializa un proceso especial, llamado proceso 0. Ordinariamente, los procesos son creados mediante una llamada a una rutina del sistema (*fork*), que funciona mediante un mecanismo de duplicación de procesos. Sin embargo, esto no es suficiente para crear el primero de ellos, por lo que el kernel asigna una estructura de datos y establece apuntadores a una sección especial de la memoria, llamada tabla de procesos, que contendrá los descriptores de cada uno de los procesos existentes en el sistema.

Luego de haber creado al proceso 0 se le hace una copia, con lo que se crea el proceso 1, que muy pronto se encargará de "dar vida" al sistema completo, mediante la activación de otros procesos que también forman parte del núcleo. Es decir, se inicia en esta forma una cadena de activaciones de procesos, entre los cuales destaca el conocido como despachador, o *scheduler*, que es el responsable de decidir cuál proceso se ejecutará y cuáles van a entrar o salir de la memoria central. (La primera vez que se llama al *scheduler*, la decisión es muy fácil ya que sólo existe el proceso 1). La ejecución del proceso 1 inmediatamente lleva a la llamada de la rutina primitiva *exec*, la cual reemplaza el código original en el proceso 1 con el código contenido en un archivo especial (`/etc/init`). A partir de este momento se conoce al proceso 1 como proceso de inicialización del sistema, `init`.

El proceso `init` es el responsable de establecer la estructura de procesos en Unix. Normalmente, es capaz de crear al menos dos estructuras distintas de procesos: el modo monousuario y el modo multiusuario. `init` comienza activando el intérprete del lenguaje de control (Shell) a la terminal principal, o consola, del sistema y dándole privilegios de "superusuario". En este modo de un solo usuario la consola permite iniciar una primera sesión, con privilegios especiales, a la vez que ninguna de las otras líneas de comunicación aceptará iniciar sesiones nuevas. El modo de un solo usuario se usa frecuentemente para revisar y reparar sistemas de archivos, y para realizar pruebas de funciones básicas del sistema y otras actividades que requieren uso exclusivo de la computadora.

Durante la operación del sistema, el proceso `init` "duerme" esperando la terminación de alguno de sus procesos hijos. Si uno de ellos termina, entonces `init` activa otro programa, llamado `getty`, empleado para atender las líneas de comunicación que por lo común están conectadas con las terminales de video. En esta forma se crea y mantiene la estructura de procesos multiusuario.

Cada proceso `getty` espera pacientemente a que alguien entre en sesión en alguna línea de comunicación. Cuando esto sucede, realiza ajustes en el protocolo de la línea y ejecuta el programa `login`, que se encarga de atender inicialmente a los nuevos usuarios. Si la clave de usuario y la contraseña proporcionadas son las correctas, `login` ejecuta el programa `shell`, que de ahí en adelante se encarga de la atención normal del usuario que se dio de alta en esa terminal.

A partir de ese momento el responsable de atender al usuario en esa terminal es el intérprete Shell, que ofrece todo un amplio conjunto de órdenes y subsistemas para la operación de la computadora, para creación y manipulación de archivos y directorios, para compilación y ejecución de programas y, en general, para mantener la comunicación entre la computadora y los usuarios.

Cuando se desea terminar la sesión hay que desconectarse de Shell (y, por tanto, de Unix), y esto se logra mediante una secuencia especial de teclas (usualmente, <CTL>-D). A partir de ese momento la terminal queda disponible para atender a un nuevo usuario.

Manejo de memoria

Dependiendo de la computadora en la que se ejecute, Unix utiliza dos técnicas de manejo de memoria: *swapping* y memoria virtual.

Lo estándar en Unix es un sistema de intercambio de segmentos de un proceso entre memoria principal y memoria secundaria, llamado *swapping*, lo que significa que se debe mover la imagen de un proceso al disco si éste excede la capacidad de la memoria principal, y copiar el proceso completo a memoria secundaria. Es decir, durante su ejecución, los procesos son cambiados de y hacia memoria secundaria conforme es necesario.

Si un proceso necesita crecer, pide más memoria al sistema operativo y se le da una nueva sección, lo suficientemente grande para acomodarlo. Entonces se copia el contenido de la sección usada al área nueva, se libera la sección antigua y se actualizan las tablas de descriptores de procesos. Si no hay suficiente memoria en el momento de la expansión, el proceso se bloquea temporalmente y se le asigna espacio en memoria secundaria. Se copia a disco y, posteriormente, se devuelve a memoria principal cuando se tenga el espacio adecuado, lo cual sucede normalmente al cabo de unos cuantos segundos.

Está claro que el proceso que se encarga de los intercambios entre memoria y disco (llamado *swapper*) debe ser especial y jamás podrá perder su posición privilegiada en la memoria central. El kernel se encarga de que nadie intente siquiera interrumpir a este proceso, del cual dependen todos los demás. Este es el proceso 0 mencionado anteriormente.

Cuando se decide traer a memoria principal un proceso en estado de "listo para ejecutar", se le asigna memoria y se copian allí sus segmentos. Entonces el proceso cargado compite por el procesador con todos los procesos cargados (es decir, es sujeto del tratamiento normal de *scheduling*). Si no hay suficiente memoria, el proceso de intercambio examina la tabla de procesos para determinar cuál puede ser interrumpido y llevado al disco.

Una pregunta que surge entonces es ¿cuál de los posibles procesos que están cargados será "congelado" y cambiado a memoria secundaria? Los procesos que se eligen primero son aquellos que están esperando eventos lentos (E/S), o que ya llevan cierto tiempo sin haberse movido al disco. La idea es tratar de repartir equitativamente las oportunidades de ejecución entre todos los procesos, tomando en cuenta sus historias recientes y sus patrones de ejecución.

Otra pregunta es ¿cuál de los varios procesos que están en disco será traído a memoria principal? La decisión se toma con base en el tiempo de residencia en memoria secundaria. El proceso más antiguo es el que se llama primero, con una pequeña penalización para los procesos grandes.

Por otro lado, cuando Unix opera en máquinas más grandes, entonces suele disponer de manejo de memoria de paginación por demanda. El tamaño de la página en Unix es de 512 bytes en algunos sistemas, y de 1024 en otros. Para reemplazo se emplea un algoritmo que mantiene en memoria las páginas usadas más recientemente

Como se mencionó en el capítulo 4, el sistema de paginación por demanda ofrece muchas ventajas en cuanto a flexibilidad y agilidad en la atención con-

currente de múltiples procesos y proporciona además memoria virtual, es decir la capacidad de trabajar con procesos de tamaño mayor que el de la memoria central. También se explicó que uno de estos esquemas en bastante complejo, y que requiere de apoyos de hardware especializado.

Manejo del procesador

En Unix, un usuario ejecuta programas en un medio llamado "proceso de usuario". Cuando se requiere una función del kernel, el proceso de usuario hace una llamada especial al sistema y entonces el control pasa temporalmente al núcleo. Para esto se requiere un conjunto de elementos de uso interno, que se describen a continuación.

Se conoce como imagen a una especie de fotografía del ambiente de ejecución de un proceso, e incluye una descripción de la memoria, valores de registros generales, status de archivos abiertos, el directorio actual, etc. Una imagen es el estado actual de una computadora virtual, dedicada a un proceso en particular.

Un proceso se define como la ejecución de una imagen. Mientras el procesador ejecuta un proceso, la imagen debe residir en memoria principal; durante la ejecución de otros procesos permanece en memoria principal a menos que la aparición de un proceso activo de mayor prioridad la obligue a ser copiada al disco, como ya se dijo.

Un proceso puede estar en uno de varios estados:

- en ejecución,
- listo para ejecutar o
- en espera.

Cuando se invoca una función del sistema, el proceso de usuario llama al kernel como subrutina. Hay un cambio de ambientes y, como resultado, se tiene un proceso del sistema. Estos dos procesos son dos fases del mismo original, que nunca se ejecutan simultáneamente.

A los procesos en ejecución se asignan tres áreas diferentes en la memoria central de la computadora: el segmento para texto, el segmento para datos y el segmento para la pila. El primero contiene el código ejecutable, que puede ser compartido (y que se compila empleando una opción especial que separa la imagen del proceso en dos: parte de variables y parte de código compartido. A esta última se le conoce como código reentrante). El segmento de texto es sólo de lectura. El segundo segmento, de datos, contiene todos los valores de las variables y datos que emplea el usuario del programa, y puede crecer dinámicamente; el tercero, el segmento para la pila, guarda la información que el sistema requiere para desactivar y activar el proceso.

Existe también una tabla de procesos que contiene una entrada por cada proceso, con los datos que el sistema requiere para cada uno: identificación, direcciones de los segmentos, información de *scheduling* y otros datos. La entrada de la tabla de procesos se asigna cuando el proceso se crea, y se libera cuando termina.

Crear un proceso requiere la inicialización de una entrada en la tabla, así como la creación de segmentos de texto y de datos. Además, es necesario modificar la tabla cuando cambia el estado del proceso o cuando recibe un mensaje de otro (para sincronización, por ejemplo). Cuando un proceso termina, su entrada en la tabla se libera y queda disponible para que otro nuevo la utilice.

En el sistema operativo Unix los procesos pueden comunicarse internamente entre sí, mediante envío de mensajes o señales. El mecanismo conocido como interconexión (*pipe*) crea un canal entre dos procesos mediante una llamada a una rutina del kernel, y se emplea tanto para pasar datos unidireccionalmente entre las imágenes de ambos, como para sincronizarlos, ya que si un proceso intenta escribir en un *pipe* ocupado, debe esperar a que el receptor lea los datos pendientes. Igual sucede para el caso de una lectura de datos inexistentes: el proceso que intenta leer debe esperar a que el proceso productor deposite los datos en el canal de intercomunicación.

Algunos procesos comunes, como Shell y `getty`, son normalmente ejecutables para varios usuarios al mismo tiempo. Cada proceso de usuario debe tener su copia de la parte de variables en la imagen del proceso (en el segmento de datos), pero la parte fija, que es el segmento de texto del programa, puede ser compartida por varios usuarios porque está formada, como se ha dicho, de código reentrante.

Entre las diferentes llamadas al sistema para manejo de procesos que existen en Unix están las siguientes, algunas de las cuales ya han sido mencionadas:

Llamadas al sistema en Unix

- `fork` (duplicar un proceso)
- `exec` (cambiar la identidad de un proceso)
- `kill` (enviar una señal a un proceso)
- `signal` (especificar la acción por ejecutar cuando se recibe una señal de otro proceso)
- `exit` (terminar un proceso)

Dentro de las tareas del manejo del procesador, explicadas en el capítulo 4, destaca la de la asignación dinámica (*scheduling*), que en Unix resuelve el *scheduler* mediante un mecanismo de **prioridades**. Cada proceso tiene asignada una prioridad; las prioridades de los procesos de usuario son menores que la prioridad más pequeña de un proceso del sistema.

Como se explicó en el capítulo de programación de sistemas, el "motor" que mantiene en movimiento un esquema de multiprogramación es el conjunto de interrupciones que genera el desempeño de los procesos, por un lado, y los constantes recordatorios que el reloj del procesador hace, indicando que se terminó la fracción de tiempo dedicada a cada proceso, por el otro.

En Unix, las interrupciones son causadas por lo que se conoce como **eventos**, entre los cuales se consideran

- La ejecución de una tarea de E/S
- La terminación de los procesos dependientes de otro

- La terminación de la fracción de tiempo de un proceso
- La recepción de una señal desde otro proceso

En un sistema de tiempo compartido se divide el tiempo en un número de intervalos o fracciones y se asigna cada una de ellas a un proceso. Unix toma en consideración además que hay procesos en espera de una operación de E/S y que ya no pueden aprovechar su fracción. Para asegurar una buena distribución del procesador entre los procesos, se calculan dinámicamente las prioridades de los procesos para determinar cuál será el proceso que se ejecutará cuando se suspenda el proceso activo actual.

El kernel asigna las prioridades iniciales. Posteriormente, se asigna una prioridad que depende de la cantidad de tiempo de procesamiento: a mayor tiempo acumulado corresponde menor prioridad. Si un proceso usa su alta prioridad para ganar procesador, entonces ésta disminuye. Si un proceso es ignorado debido a su baja prioridad, se ve aumentada. Los procesos que acaban de entrar a memoria principal (por el mecanismo de *swapping*), y los que están en espera de un evento de E/S, tienen prioridad alta.

Manejo de entradas y salidas

El sistema de entrada/salida se divide en dos sistemas complementarios: sistema de E/S estructurado **por bloques**, y sistema de E/S **por caracteres**. El primero se emplea para el manejo de discos y cintas magnéticos, y emplea bloques de tamaño fijo (512 ó 1024 bytes) para leer o escribir. El segundo se emplea para la atención a las terminales, líneas de comunicación e impresoras, y funciona byte por byte.

En general, Unix emplea programas especiales (escritos en C) conocidos como manejadores (*drivers*) para atender a cada familia de dispositivos de E/S. Los procesos se comunican con los dispositivos mediante llamadas a su manejador. Además, desde el punto de vista de los procesos, los manejadores aparecen como si fueran archivos en los que se lee o escribe, logrando con esto una gran homogeneidad y elegancia en el diseño.

Cada dispositivo se estructura internamente mediante descriptores llamados **número mayor, número menor** y **clase** (de bloque o de caracteres). Para cada clase hay un conjunto de entradas, en una tabla, que apuntan a los manejadores de los dispositivos. El número mayor se usa para asignar el manejador correspondiente a una familia de dispositivos. El número menor del dispositivo pasa al manejador como un argumento, y éste lo emplea para tener acceso a uno de varios dispositivos físicos semejantes.

Las rutinas que el sistema emplea para ejecutar operaciones de E/S están diseñadas para eliminar las diferencias entre los dispositivos y los tipos de acceso. No existe distinción entre acceso aleatorio y secuencial, ni hay un tamaño de registro lógico impuesto por el sistema. El tamaño de un archivo ordinario está determinado por el número de bytes escritos en él; no es necesario predeterminar el tamaño de un archivo.

El sistema mantiene una lista de áreas de almacenamiento temporal (*buffers*), asignados a los dispositivos de bloques. El kernel usa estos buffers con el objeto de reducir el tráfico de E/S. Cuando un programa solicita una transferencia, se busca primero en los buffers internos para ver si el bloque que se requiere ya se encuentra en la memoria principal (como resultado de una operación de lectura anterior). Si es así, entonces no será necesario realizar la operación física de entrada o salida.

Existe todo un mecanismo de manipulación interna de buffers (y otro de manejo de listas de bytes), que se requieren para controlar el flujo de datos entre los dispositivos de bloques (y de caracteres) y los programas que los requieren.

Por último, y debido a que los manejadores de los dispositivos son programas escritos en el lenguaje C, es relativamente fácil reconfigurar el sistema para ampliar o eliminar dispositivos de E/S en la computadora, así como para incluir tipos nuevos.

Manejo de archivos y manejo de información

Cómo ya se describió, la estructura básica del sistema de archivos es jerárquica, lo que significa que los archivos no están almacenados en un nivel sino en varios. Se puede tener acceso a cualquier archivo mediante su trayectoria, que especifica su posición absoluta en la jerarquía, y los usuarios pueden cambiar su directorio actual a cualquier posición. Existe un mecanismo de protección para evitar accesos no autorizados.

La distinción entre un directorio y un archivo ordinario es que el sistema se reserva el derecho de alterar el contenido de los primeros, y que el usuario sólo los puede manipular mediante las órdenes ya mencionadas (`mkdir`, `rmdir`, etc.).

Los directorios contienen información para cada archivo, que consiste en su nombre y en un número que el kernel utiliza para manejar la estructura interna del sistema de archivos, conocido como el **nodo-i**. Hay un nodo-i para cada archivo, que contiene información de su dirección en el disco, su longitud, los modos de acceso, las fechas de acceso, el autor, etc. Existe, además, una tabla de descriptores de archivos, que es una estructura de datos residente en el disco magnético, a la que se tiene acceso mediante el sistema de E/S por bloques ya mencionado.

Pueden existir varios sistemas de archivos independientes, y una misma unidad de disco magnético puede contener varios de ellos. Cada uno de los sistemas de archivos está dividido internamente en cuatro secciones o particiones lógicas. La primera (conocida como bloque 0) se reserva para procedimientos de *bootstrap*. La segunda (con identificador 1) contiene lo que se conoce como el "superbloque", que almacena un descriptor de la estructura de todo el sistema de archivos. La tercera área (identificador 2) es una lista de definiciones de archivos llamada lista-i (ésta es la tabla de archivos), en la que cada definición de archivo es una estructura de 64 bytes: el nodo-i.

El desplazamiento de un nodo-i particular dentro de la lista-i es el **número-i** de un archivo, y actúa como su índice. La combinación del nombre del dispositivo y su número en esta lista sirven para identificar en forma única todo archivo. El final de cada sistema de archivos (la cuarta área) se usa para almacenar el contenido de los archivos, y es la sección de mayor tamaño.

El control del espacio libre en el disco se mantiene mediante una lista ligada de bloques disponibles. Cada bloque contiene la dirección en disco del siguiente bloque en la cadena. El espacio restante contiene las direcciones de grupos de bloques del disco que se encuentren libres. De esta forma, con una operación de E/S, el sistema obtiene un conjunto de bloques libres y un apuntador para obtener más.

Un nodo-i contiene 13 espacios para direcciones (de 4 bytes de longitud cada uno), en los cuales se encuentra la localización de un archivo. Las primeras 10 direcciones apuntan directamente a los primeros 10 bloques del archivo. Esto es suficiente para describir un archivo de hasta 10 × 512 bytes de longitud. Si el archivo es más grande; entonces se emplea la dirección 11, que apunta a un bloque que contiene hasta 128 direcciones de bloques adicionales. Es decir, el acceso a un byte situado entre la posición 5121 y la posición 70 656 (o sea, 512 × (10 + 128)) requiere de un acceso indirecto adicional para averiguar su posición exacta. Si es necesario un archivo aun mayor, entonces la dirección 12 apunta a un bloque de doble indirección con 128 bloques indirectos, donde cada uno apunta a 128 bloques del archivo. Por último, la dirección 13 apunta a un bloque de triple indirección, lo que permite un tamaño máximo (en la versión conocida como Unix III) para un archivo de 1 082 201 087 bytes: $[(10 + 128 + 128^2 + 128^3) \times 512]$. En las versiones posteriores (Unix V), este número se incrementa porque, ahí, los bloques con los que se maneja el disco magnético son de 1024 bytes de longitud, no de 512.

Las operaciones de E/S en archivos se llevan a cabo con la ayuda de la correspondiente entrada del nodo-i en la tabla de archivos del sistema. El usuario normalmente desconoce los nodos-i y los números-i porque las referencias se hacen por el nombre simbólico de la trayectoria. Los procesos emplean internamente funciones primitivas (llamadas al sistema) para tener acceso a los archivos; las más comunes son `open`, `creat`, `read`, `write`, `seek`, `close` y `unlink`, que se comentan más adelante.

Toda esta estructura física se maneja "desde afuera" mediante la filosofía jerárquica de archivos y directorios ya mencionada, y en forma totalmente transparente para el usuario. Desde su punto de vista, hay tres clases de archivos: **ordinarios, directorios** y **especiales**.

Un archivo ordinario se usa para almacenar información. Puede contener un programa, el texto de un documento, los registros de una compañía, o cualquier otro tipo de información que se desee procesar en una computadora. Existen dos tipos de archivos ordinarios: archivos de texto y archivos binarios. El sistema no presupone una estructura particular en un archivo, sino que deja a los programas de los usuarios la tarea de manejar y controlar su estructura. Desde el punto de vista de Unix, un archivo no es más que un conjunto de bytes.

Los directorios proporcionan la liga entre los nombres de los archivos y los archivos mismos; es decir, determinan una estructura en el sistema de archivos. Se deja como responsabilidad de los usuarios la formación de su estructura arborescente en particular, pero el sistema operativo es el único que puede alterar internamente el contenido de un directorio.

En los archivos especiales reside el medio de control sobre los dispositivos de E/S. Cada dispositivo está asociado con al menos uno de esos archivos, que se leen y se escriben como si fueran un archivo ordinario de disco, pero en realidad las solicitudes de lectura y escritura activan el dispositivo asociado. Los archivos especiales no contienen información, sino que son utilizados para proporcionar un canal conveniente para los mecanismos de E/S.

Existe un directorio dedicado a contener los archivos especiales (`/dev`). Para grabar información sobre una cinta magnética, por ejemplo, se escribe en el archivo `/dev/mt`. Existen archivos especiales para cada línea de comunicación, cada disco, cada unidad de cinta y para la memoria principal física. Existe, claro, un mecanismo de protección para evitar el acceso indiscriminado.

Las ventajas de tratar a los dispositivos E/S de esta manera son múltiples: un archivo y un dispositivo de E/S se vuelven muy similares; los nombres de archivos y de dispositivos tienen la misma sintaxis y significado, así que a un programa que espera un nombre de archivo como parámetro puede dársele un nombre de dispositivo (con esto se logra interacción rápida y fácil entre procesos de alto nivel); por último, los archivos especiales están sujetos al mismo mecanismo de protección de los archivos regulares.

El **modo de protección** consiste en asignar a cada archivo el número único de identificación de su dueño, junto con 9 bits de protección que especifican permisos de lectura, escritura y ejecución para el propietario, para otros miembros de su grupo (definido por el administrador del sistema), y para el resto de los usuarios. Antes de cualquier acceso se verifica su validez consultando estos bits, que residen en el nodo-i de todo archivo. Además de lo anterior existen otros tres bits que se emplean para manejos especiales relacionados con la clave del superusuario.

Las principales primitivas para acceso interno al sistema de archivos, ya mencionadas, son las que siguen:

- `open` (convierte un nombre simbólico del sistema de archivos —trayectoria— en una entrada a la tabla de nodos-i).
- `creat` (crea una nueva entrada en la tabla de nodos-i).
- `read` y `write` (transferencia de un número determinado de bytes entre un archivo y la memoria).
- `seek` (permite acceso aleatorio dentro de un archivo)
- `close` (libera las estructuras creadas por `open` y `creat`)
- `unlink` (elimina un archivo del sistema)

El acceso inicial a un archivo es mediante las primitivas `open` o `creat`. Ambas devuelven un número conocido como **descriptor de archivo**, que sirve como conector entre el archivo y las llamadas de E/S del programa (un número negativo indica un error de algún tipo).

Otra característica de Unix es que no requiere que el conjunto de sistemas de archivos resida en un mismo dispositivo. Es posible definir uno o varios sistemas "desmontables", que residen físicamente en unidades de disco diversas. Existe una orden (`mkfs`) que permite crear un sistema de archivos adicional, y una llamada al sistema (`mount`) con la que se añade (y otra con la que se quita) uno de ellos al sistema de archivos global.

El control de las impresoras de una computadora que opera con el sistema operativo Unix es mediante un subsistema (*SPOOL*) que se encarga de coordinar los pedidos de impresión de múltiples usuarios. Existe un proceso del kernel (`/usr/lib/lpd`) que periódicamente revisa las colas de servicio de las impresoras para detectar la existencia de pedidos e iniciar entonces las tareas de impresión. Este tipo de procesos, que son activados en forma periódica por el núcleo del sistema operativo, reciben en Unix el nombre de *daemons* (duendes), tal vez porque se despiertan y aparecen sin previo aviso. Otros son `/etc/cron`, que se encarga de activar procesos en tiempos previamente determinados por el usuario, o `/etc/update`, que periódicamente escribe los contenidos de los buffers de memoria en el disco magnético.

Lenguaje de control del sistema operativo

Entre los rasgos distintivos de Unix está el lenguaje de control que emplea, Shell. Es importante analizar dos funciones más de Shell, llamadas redireccionamiento e interconexión.

Dentro de los procesos que suceden cuando un usuario se da de alta, destaca el hecho de que se crea una copia del programa intérprete Shell y se le asigna para su atención constante. Cuando se ejecuta un programa del usuario, Shell analiza la línea donde está escrita la orden de ejecución y la interpreta; para realizar esta tarea crea un proceso hijo. En esa línea se puede especificar, además, qué archivo debe usarse como entrada y cuál como salida, y eso se conoce como **redireccionamiento** de entrada/salida.

Asociado con cada proceso hay un conjunto de descriptores de archivo numerados 0, 1 y 2, que se utilizan para todas las transacciones entre los procesos y el sistema operativo. El descriptor de archivo 0 se conoce como la entrada estándar; el descriptor de archivo 1, como la salida estándar, y el descriptor 2, como el error estándar. Todos están normalmente asociados con la terminal de video, pero debido a que son incialmente establecidos por Shell, es posible reasignarlos.

Una parte de la orden que comience con el símbolo < se considera como el nombre del archivo que será abierto por Shell y que se asociará con la entrada estándar; en su ausencia, la entrada estándar se asigna a la terminal. En forma similar, un archivo cuyo nombre está precedido por el símbolo > recibe la salida estándar de las operaciones.

Cuando Shell interpreta la orden

```
califica < examem > result
```

llama a ejecución al programa `califica` (que ya debe estar compilado y listo para ejecutar) y crea un proceso hijo que detecta la existencia de un archivo que toma el lugar de la entrada estándar, y de otro que reemplaza a la salida estándar. Luego se encarga de pasar como datos de lectura los contenidos del archivo `examen` recién abierto (que debe existir previamente) al programa ejecutable. Conforme el programa produce datos como salida, éstos se guardan en el archivo `result` que Shell crea en ese momento.

En el capítulo 5, al hablar de las gramáticas y los reconocedores de tipo 3 (también conocidos como regulares), se dijo que tienen múltiples aplicaciones en el manejo de lenguajes, entre las que destacan las llamadas expresiones regulares. Con una sola expresión regular se puede hacer referencia a un conjunto ilimitado de nombres con estructura lexicográfica similar, y esto es aprovechado por Shell para dar al usuario facilidades expresivas adicionales para el manejo de los nombres de los archivos. Así, por ejemplo, el nombre `carta*` se refiere a todos los archivos que comiencen con el prefijo `carta` y que sean seguidos por cualquier subcadena, incluyendo la cadena vacía; por ello, si se incluye el nombre `carta*` en alguna orden, Shell la aplicará a los archivos `carta` , `carta1` , `carta2` , y a cualquier otro que cumpla con esa especificación abreviada. En general, en cualquier lugar donde se emplee un nombre o una trayectoria, Shell permite utilizar una expresión regular que sirve como abreviatura para toda una familia de ellos, y automáticamente repite el pedido de atención para cada uno de los componentes. Existen además otros caracteres especiales que Shell reconoce y emplea para el manejo de expresiones regulares, dando al lenguaje de control de Unix aun mayor potencia y capacidad expresiva.

Por otro lado, en Unix existe la posibilidad de ejecutar programas sin tener que atenderlos en forma interactiva, sino simulando paralelismo (es decir, atender concurrentemente varios procesos de un mismo usuario). Esto se logra agregando el símbolo `&` al final de la línea en la que se escribe la orden de ejecución. Como resultado, Shell no espera que el proceso hijo termine de ejecutar (como haría normalmente), sino que regresa a atender al usuario inmediatamente después de haber creado el proceso asincrónico, simulando en esta forma procesamiento por lotes (*batch*). Para cada proceso de éstos, Shell informa, además, el número de identificación, por lo que el usuario puede cancelarlo posteriormente, si fuera necesario (mediante la orden `kill`), o averiguar el avance de ejecución (mediante la orden `ps` —*process status*—).

La comunicación interna entre procesos (es decir, el envío de mensajes con los que los diversos procesos se sincronizan y se coordinan) ocurre mediante el mecanismo de interconexiones (*pipes*) ya mencionado, que conecta la salida estándar de un programa a la entrada estándar de otro, como si fuera un conducto con dos extremos, cada uno de los cuales está conectado a su vez a un proceso distinto. Desde Shell se puede emplear este mecanismo con el símbolo `|` en la línea donde se escribe la orden de ejecución.

Así, en el ejemplo

```
( califica < tareas | sort > lista ) &
```

se emplean las características de interconexión, redireccionamiento y asincronía de procesos para lograr resultados que en otros sistemas operativos son bastante más difíciles de obtener. Aquí se pide que, en forma asincrónica (es decir, dejando que la terminal siga disponible para atender otras tareas del mismo usuario) se ejecute el programa `califica` para que lea los datos que requiere del archivo `tareas`; al terminar, se conectará con el proceso `sort` (es decir, pasará los resultados intermedios) para que continúe el procesamiento y se arreglen los resultados en orden alfabético; al final de todo esto, los resultados quedarán en el archivo `lista`.

Con esta otra orden, por ejemplo, se buscan las apariciones de todos los renglones que contengan las palabras "contrato" o "empleado" en todos los archivos en disco cuyos nombres comiencen con la letra "E" (lo cual se denota mediante una expresión regular). Se hace uso de una función llamada `egrep`, especial para manejo de patrones y combinaciones de expresiones regulares dentro de archivos:

```
egrep -n 'contrato' ¦'empleado' E*
```

Los resultados aparecen así:

```
Emple1:5: en caso de que un empleado decida hacer uso de la facilidad,
Emple1:7: y el contrato asi lo considere: las obligaciones de la
Emple2:9: Clausula II: El contrato colectivo de trabajo especi-
Emple2:15: Fraccion III: El empleado tendra derecho, de acuerdo
```

(El tercer renglón, por ejemplo, muestra el noveno renglón del archivo `Emple2`, que contiene una de las palabras buscadas).

Como Unix fue diseñado para servir de entorno en las labores de diseño y producción de programas, ofrece —además de su filosofía misma— un rico conjunto de herramientas para la creación de sistemas complejos, entre las que destaca el subsistema `make`. `make` ofrece una especie de lenguaje muy sencillo, con el que el programador describe las relaciones estructurales entre los módulos que configuran un sistema completo, para que de ahí en adelante `make` se encargue de mantener el sistema siempre al día. Es decir, si se modifica algún módulo, se reemplaza o se añade otro, las compilaciones individuales, así como las cargas y ligas a que haya lugar, serán efectuadas en forma automática por esta herramienta. Con una sola orden, entonces, es posible efectuar decenas de compilaciones y ligas predefinidas entre módulos, y asegurarse de que en todo momento se tiene la última versión de un sistema, ya que también se lleva cuenta automática de las fechas de creación, modificación y compilación de los diversos módulos, con lo que `make` se convierte en una herramienta casi indispensable al desarrollar aplicaciones que requieren decenas de programas que interactúan entre sí o que mantienen relaciones jerárquicas entre ellos.

Otras herramientas interesantes son `ar`, un programa para crear y mantener bibliotecas de programas (que serán luego utilizadas por otros programas

para efectuar las funciones ya definidas sin tener que duplicar el código); `awk`, un lenguaje para reconocimiento de patrones y expresiones regulares (es decir, generadas por una gramática regular o de tipo 3), útil para extraer información de archivos en forma selectiva; `lex`, un generador de analizadores lexicográficos, y `yacc`, un compilador de compiladores. Estos dos últimos se emplean como herramientas en la creación de compiladores y procesadores de lenguajes.

Comentarios finales

Un sistema operativo es mucho más que un amplio conjunto de programas; representa, de hecho, la *forma* que tendrá una computadora ante sus usuarios. Y es aquí donde Unix es un tanto especial, pues fue diseñado originalmente con una filosofía muy clara y explícita: servir como marco de referencia para el desarrollo de software. Esta marca de origen explica, al mismo tiempo, el gran éxito de Unix entre la comunidad académica y computacional y su relativamente menor penetración y popularidad en el mercado del procesamiento de datos y la informática comercial. De hecho, el objetivo primario de una computadora (y, por ende, del sistema operativo que la hace ser lo que es) en el entorno comercial o de producción es precisamente servir como vehículo para la explotación de sistemas ya creados. Por tanto, al usuario de un sistema tal le preocupa más la facilidad de operación que la posibilidad de crear estructuras computacionales elegantes o complejas.

La corta historia de la computación ha mostrado que en este campo del quehacer humano (como en todos los demás) no hay panaceas ni soluciones universales. Aquí han existido también los inevitables intentos por definir la realidad de acuerdo con intereses particulares o de mercado, y al paso de los años los hemos visto fracasar. En el campo de los lenguajes de programación, por ejemplo, se ha propuesto que tal o cual lenguaje es el adecuado (Algol y PL/I han sido ejemplos de esta megalomanía, así como más recientemente lo es Ada), y el futuro próximo depara más revelaciones y sorpresas de este tipo.

La filosofía que subyace al diseño de Unix apunta claramente hacia un campo de aplicaciones creativas, pero que requieren conocimiento especializado previo. No es, en efecto, un sistema oscuro o intencionalmente difícil, pero tampoco fue creado pensando en usuarios casuales o poco interesados. Para este tipo de mercado existen en muchas máquinas de Unix "frentes amigables" que guían al usuario mediante menús y pantallas de ayuda, por lo que tampoco quedan excluidos de su espectro de aplicaciones.

Sin embargo, los que emplean Unix como herramienta y entorno de creación de programas y sistemas encuentran en él un campo extremadamente fértil (hasta podríamos decir exuberante) para sus esfuerzos. Esto se debe, como se ha dicho, a la filosofía de herramientas de software con la cual fue creado. El reconocimiento de la comunidad académica internacional también llegó ya; la prestigiada *Association for Computing Machinery* otorgó a Den-

nis Ritchie y Ken Thompson el premio Alan Turing de 1983, por su labor en el desarrollo de Unix*.

De hecho, Unix no es un sistema operativo monolítico, como casi todos los demás, sino que está compuesto de un pequeño núcleo y un conjunto (que a veces parece casi ilimitado) de rutinas y operadores, que literalmente crean una verdadera atmósfera que envuelve a la programación y diseño de sistemas. Es más, el campo conocido como ingeniería de software —esto es, la creación de programas y sistemas con un método científico y no basado en el método de ensayo y error— ve en Unix casi la culminación de sus expectativas, puesto que el diseñador de sistemas se rodea de herramientas de todo tipo, que van desde comparadores de archivos y contadores de palabras hasta subsistemas completos para la generación de reconocedores de lenguajes.

Todo esto, además, está ligado al hecho de que Unix es un sistema operativo relativamente caro en recursos: requiere de un sistema de disco rígido rápido y eficaz; requiere de velocidad del procesador central; requiere manejadores eficaces de entrada y salida. No todo esto está accesible en las computadoras personales, ni éstas fueron diseñadas con ello en mente. No es de extrañar, pues, que Unix sea menos popular que el sistema estándar en computadoras personales (MS-DOS ahora y OS/2 en un futuro), o que su filosofía de uso sea otra.

Nada de lo anterior, por supuesto, impide que Unix sea un vehículo óptimo para la productividad, tanto operativa como de diseño y, de hecho, cuando uno ha trabajado con Unix, ya no desea sentirse desprotegido.

Funciones principales incluidas en Unix

Se muestra un compendio de las funciones más importantes que incluye el sistema operativo Unix (aunque existen muchas más). En general, cada una de las órdenes cuenta también con un conjunto de opciones extra de procesamiento.

Funciones para control de usuarios

`login`	Manejo de la conexión de un nuevo usuario y preparación de su entorno de trabajo en la terminal.
`newgrp`	Cambios de grupos de usuarios.
`passwd`	Manejo y cambio de las claves secretas de acceso.

* Este premio es considerado el máximo reconocimiento a la calidad académica o profesional en el campo de la computación en el mundo. Los premiados pronuncian una conferencia en la ceremonia de aceptación, y una versión adaptada se publica después en la revista oficial de la asociación, la ya muchas veces citada *Communications of the ACM*. Los artículos de Ritchie y Thompson aparecieron en el número de agosto de 1984. La editorial Addison-Wesley publicó en 1987 el libro *ACM Turing Award Lectures: The First Twenty Years*, que contiene los artículos de los premiados entre 1966 y 1985, y se trata prácticamente de un directorio de los principales investigadores en computación del mundo, ya que aparecen, entre otros, los nombres de Knuth, Dijkstra, Backus, Hoare, Wirth, Wilkes, Minsky, McCarthy, Simon, Rabin, Iverson y Codd.

Manejo de terminales

`stty`	Determinación de opciones específicas de la terminal.
`tab`	Manejo generalizado de posiciones de tabuladores.

Manejo de archivos y directorios

`cat`	Concatenación de uno o varios archivos. Despliegue en pantalla.
`cd`	Cambio de directorio de trabajo.

`chmod`
`chown`
`chgrp`
	Cambio de permisos y facilidades de acceso a archivos y directorios.

`cmp`	Comparación de archivos e informe de diferencias.
`cp`	Copia de archivos o grupos de ellos.
`dd`	Traductor automático de formatos entre archivos.
`find`	Búsquedas estructuradas de archivos, de acuerdo con criterios de fecha de creación, patrones de letras en los nombres o combinaciones lógicas.
`ln`	Liga de archivos entre sí.
`mkdir`	Creación de nuevos directorios.
`mv`	Cambio de nombres de archivos o directorios.
`pack`	Compactación de archivos para ahorrar espacio en disco.
`pcat`	Función idéntica a `cat`, para archivos compactados.
`pr`	Impresión de archivos paginados y con fecha.
`rm`	Eliminación de archivos.
`split`	Fraccionamiento de un archivo en partes.
`tail`	Despiegue de las últimas *n* líneas de un archivo.
`unpack`	Regreso de un archivo compactado a su estado original.

Ejecución de programas

`echo`	Escritura de mensajes por pantalla.
`kill`	Terminación de la ejecución de un proceso.
`line`	Lectura de órdenes por pantalla.
`nice`	Cambio de las prioridades de ejecución de órdenes y procesos.
`sh`	Intérprete del lenguaje de control. Manejo de argumentos. Redireccionamiento de entradas/salidas. Interconexión de procesos (*pipes*). Inicio de procesos por lotes (*batch*). Manejo de listas de argumentos y de variables de control.

sleep	Suspensión de la ejecución de una orden durante un tiempo especificado en segundos.
tee	Paso de datos entre procesos en ejecución y copia de los resultados obtenidos.
test	Prueba y uso de condiciones en Shell.
wait	Espera de la terminación de un proceso asincrónico

Funciones para control de status

date	Informe de la fecha, que el sistema calcula automáticamente.
df	Informe de la cantidad de espacio disponible en los sistemas de archivos.
du	Despliegue de un resumen de la utilización del disco.
file	Determinación del tipo de información que contiene un archivo.
ls	Lista en orden de los nombres de los archivos en el sistema del usuario, con diversos grados de detalle.
ps	Informe de la actividad de los procesos del sistema en ejecución, activos y suspendidos.
pwd	Identificación del directorio actual de trabajo.
tty	Identificación de la terminal en la que se está trabajando.
who	Informe de los usuarios conectados al sistema. Informe de una historia de las conexiones anteriores.

Mantenimiento y respaldos

cpio	Operación de los dispositivos de almacenamiento masivo.
dump	Respaldo automático, selectivo y total, del sistema de archivos de la computadora.
fsck	Despliegue y reparación del sistema de archivos de la computadora, sus ligas, bloques usados, consistencia, tamaño, etcétera.
mount	Asignación de un sistema de archivos a un disco.
restor	Recuperación del sistema de archivos.
su	Asignación temporal de permisos privilegiados de acceso al sistema de archivos.
sync	Terminación de las operaciones de E/S pendientes.
tar	Manejo de la unidad de cinta magnética.
unmount	Cancelación de la asignación hecha con mount.

Funciones para impresión

lp	SPOOLer para control de impresoras y pedidos de impresión.
lpr	Manejo de pedidos de impresión.

`lpstat`	Despliegue de información sobre el sistema de colas de impresión.
`pr`	Paginación de un archivo.

Manejo de información

`awk`	Lenguaje para procesamiento de patrones en textos.
`calendar`	Servicio automático de recordatorios y fechas.
`comm`	Identificador de líneas comunes en dos archivos ordenados.
`diff`	Comparador de archivos e informe de diferencias.
`grep`	Despliegue de los renglones de un archivo que satisfacen criterios de reconocimiento de patrones.
`join`	Combinación de archivos con registros con llaves idénticas.
`sort`	Ordenamientos de archivos ASCII, con múltiples opciones.
`tr`	Transliterador de caracteres, de acuerdo con convenciones definidas por el usuario.
`uniq`	Informe de líneas duplicadas dentro de un archivo.

Auditoría del sistema e informe de actividades

`accdisk`	Informe de actividades de uso del disco magnético.
`accton`	Inicio de operación del sistema de auditoría interno.
`acctprcq`	Informe de actividades por proceso.
`sag`	Informe gráfico de la actividad del sistema en un periodo cualquiera.

Facilidades de comunicaciones

`cu`	Llamado a otro sistema Unix. Interfaz automática con otra computadora remota.
`mail`	Envío de mensajes a uno o varios usuarios. (Sistema de correo electrónico)
`mesg`	Control sobre los mensajes recibidos en una terminal.
`news`	Despliegue de la información del día.
`uucp`	Transferencia de archivos entre sistemas Unix.
`uulog`	Espera automática hasta lograr la conexión remota.
`uuname`	Informe de estadísticas de uso remoto.
`uupick`	Definición y control de subredes Unix.
`unstat`	Transferencias entre dos máquinas remotas.
`wall`	Mensajes a todos los usuarios, por parte del administrador del sistema.
`write`	Comunicación directa entre terminales.

Herramientas de desarrollo de programación

`adb`	Depurador interactivo. Desensamblador. Alteración de archivos objeto.
`ar`	Creación y mantenimiento de bibliotecas de programas en cinta magnética.
`as`	Ensamblador.
`ld`	Editor-ligador de uso general.
`library`	Bibliotecas comunes de tiempo de ejecución.
`lorder`	Manejo y ordenamiento de archivos objeto para ser cargados.
`make`	Sistema general para control y mantenimiento de programas, rutinas y módulos. Manejo automático de las interdependencias entre módulos.
`od`	Despliegue de códigos objeto en forma octal, hexadecimal, decimal y ASCII.
`prof`	Construcción de una tabla de estadísticas de llamadas a funciones, rutinas y tiempos de ejecución (*profiler*).
`size`	Informe de los requerimientos de memoria de archivos objeto.
`strip`	Minimización del espacio requerido por un archivo objeto.
`time`	Ejecución de una orden del sistema y reporte de los tiempos de proceso y ejecución.

Lenguaje C

`cb`	"Embellecedor" de programas en C.
`cc`	Compilador, ligador y cargador del lenguaje C. Macroprocesador integrado.
`lint`	Verificador sintáctico/semántico para el lenguaje C.

Otros lenguajes algorítmicos integrados

`bc`	Interfaz tipo lenguaje C para la calculadora `dc`.
`bs`	Intérprete y compilador para un lenguaje, que comparte características de SNOBOL4, BASIC y C.
`dc`	Calculadora programable interactiva de precisión aritmética ilimitada y manejo de múltiples bases numéricas.
`snobol`	Intérprete y compilador del lenguaje SNOBOL.

Macroprocesamiento

`m4`	Macroprocesador de propositos generales.

Compiladores de compiladores

`lex`	Sistema generador de analizadores lexicográficos.
`yacc`	Sistema generador de analizadores sintácticos LR(1).

Preparación de documentos

`troff`	
`nroff`	Sistemas completos de procesamiento de palabras y tipografía computarizada.
`deroff`	Eliminación de las órdenes para el procesador `troff` en un texto.
`ed`	Editor interactivo de línea, guiado por gramática regular; incluye reconocimiento de patrones.
`eqn`	Preprocesador para diseño de expresiones matemáticas en `troff`.
`mm`	Sistema de macros para `nroff` y `troff`.
`mmcheck`	Verificación de los documentos a ser procesados por `eqn` y `mm`.
`ptx`	Creación de tablas de índices permutados.
`sed`	Igual que `ed`, pero para archivos de tamaño ilimitado.
`spell`	Sistema de comparación automática de palabras de un texto contra un diccionario, para encontrar errores mecanográficos.
`tbl`	Preprocesador para manejo y diseño de tablas en `troff`.
`vi`	Editor de pantalla, con facilidades integradas para creación de programas en lenguaje C.

Manejo de gráficas

`graph`	Producción de una gráfica a partir de sus coordenadas.
`spline`	Ajuste gráfico de curvas por métodos matemáticos.

Referencias sobre Unix

En el capítulo 4 se mencionó un libro avanzado ([BACM86]) que describe tanto la filosofía de diseño como el funcionamiento interno del sistema operativo Unix. Aquí se relacionan varios textos y guías de propósito general.

Anderson, Gail y Paul Anderson, *The Unix C Shell Field Guide*, Prentice Hall, New Jersey, 1986.

Libro dedicado al manejo de una versión avanzada del lenguaje de control del sistema operativo, llamada C Shell. Muchas instalaciones de Unix cuentan con este intérprete, desarrollado en la Universidad de California en Berkeley, que presenta ventajas sobre el que se emplea normalmente, conocido como Bourne Shell, o simplemente Shell.

Blackburn, Lawrence y Marcus Taylor, *UNIX. Guía de bolsillo*, Fondo Educativo Interamericano, México, 1986.
Traducción de un resumen de las principales características del sistema operativo, presentadas en forma concisa en un pequeño volumen diseñado para ser colocado junto a la terminal de video. Tiene breves secciones sobre los programas de servicios generales, manejo de archivos, manejo del editor, el lenguaje de control Shell, y notas para el administrador del sistema.

Christian, Kaare, *The Unix Operating System*, Wiley, Nueva York, 1983.
En la primera parte de este libro se describe la operación de las principales características y funciones de Unix y, por ello, no se diferencia mucho de otros libros (aunque hay que decir que éste fue de los primeros). La segunda parte, de temas avanzados, trata con el lenguaje de control Shell, con el compilador de C, con el generador de analizadores lexicográficos, `lex`, y con el compilador de compiladores, `yacc`, además de describir el kernel y las llamadas al sistema. Al final incluye un resumen del manual completo del sistema operativo.

Fiedler, David y Bruce Hunter, *Unix System Administration*, Hayden, New Jersey, 1986.
Interesante libro que muestra, en términos sencillos y operativos, sin recurrir a mucha teoría, el funcionamiento interno de Unix y la manera de controlarlo, desde el punto de vista del administrador responsable de la computadora. Tiene capítulos que tratan desde el mantenimiento de usuarios y sus archivos, así como sobre respaldos, seguridad en el sistema, impresoras y terminales, hasta manejo de modems y telecomunicaciones.

Kernighan, Brian y Rob Pike, *El entorno de programación Unix*, Prentice-Hall Hispanoamericana, México, 1987.
Traducción de un libro en el que dos de los diseñadores originales de Unix explican la manera de usarlo como apoyo para las labores de creación de programas y sistemas complejos. Comienza con una descripción del sistema de archivos y luego dedica varios capítulos a la producción de programas y "filtros" en Shell; después de dos capítulos más, en los que mediante ejemplos se describe el uso de las bibliotecas estándar de E/S y de algunas llamadas al kernel, culmina con el desarrollo minucioso de un sistema completo (la simulación de una calculadora avanzada), empleando para ello prácticamente todas las herramientas avanzadas de Unix, como `lex`, `yacc` y `make`. Ade-

más, tiene un capítulo sobre creación de documentos. Se trata de un libro avanzado que presupone amplios conocimientos del lenguaje C.

McGilton, Henry y Rachel Morgan, *Introducing the Unix System*, McGraw-Hill, Nueva York, 1983.
Amplio libro que explica la operación del sistema operativo, desde el punto de vista del usuario. Dedica más de 250 páginas al procesamiento de palabras y a la preparación de documentos. Incluye una sección final sobre administración del sistema.

Ritchie, Dennis y Ken Thompson, "The UNIX Time-Sharing System", *Communications of the ACM*, julio, 1974.
Éste es el breve artículo original de los creadores de Unix. Describe las principales características de su diseño y dedica algunos párrafos a los siguientes temas, en ese orden: requerimientos de hardware (para una computadora PDP-11/45), el sistema de archivos, implantación del sistema de archivos, procesos e imágenes, Shell, modos de terminación, perspectivas, y estadísticas de uso.

Silvester, Peter, *The Unix System Guidebook: an Introductory Guide for Serious Users*, Springer-Verlag, Nueva York, 1984.
Aunque el subtítulo resulta ligeramente amenazador ("para usuarios serios"), el libro es una breve introducción a las facilidades y modos de funcionamiento interno del sistema operativo, tratado todo en un nivel intermedio; no se requiere que el lector posea conocimientos profundos de computación.

Sobell, Mark, *Guía práctica para el sistema operativo Unix*, Addison-Wesley Iberoamericana, México, 1987.
Traducción de un accesible libro sobre la operación del sistema Unix, dirigido a usuarios sin experiencia. Tiene capítulos sobre manejo general, sobre el sistema de archivos, sobre Shell y sobre los diversos subsistemas para procesamiento de textos. Dedica más de 150 páginas a describir las principales órdenes de Unix: uso, opciones y argumentos requeridos, y muestra varios ejemplos para cada uno.

Thomas, Rebecca y Jean Yates, *A User Guide to the Unix System*, Osborne/McGraw-Hill, Berkeley, 1982.
Libro de carácter introductorio para el uso del sistema operativo. Contiene gran cantidad de ejemplos sencillos del uso normal, e incluye un capítulo final con datos y direcciones de proveedores (principalmente en Estados Unidos) de sistemas de diversos tipos que funcionan con Unix, como software administrativo, bases de datos, procesadores de textos, compiladores, etcétera. Existe traducción al español.

Wood, Patrick y Stephen Kochan, *Unix System Security*, Hayden, New Jersey, 1985.

En la operación de los centros de cómputo, los aspectos de seguridad y privacidad de la información son muy importantes, y este libro dedica casi 300 páginas a describir cómo mantener estas condiciones, desde el punto de vista de los usuarios, los programadores y el administrador del sistema. Tiene también un capítulo dedicado a los aspectos de seguridad en las redes Unix y varios apéndices con texto de programas especiales para manejo de protecciones.

Apéndice B: el lenguaje C

C es un lenguaje de programación de alto nivel que permite, de manera completa y conveniente, la expresión de algoritmos de programación estructurada, así como la codificación de aplicaciones especializadas de programación de sistemas, de tal forma que actualmente muchos compiladores, monitores, sistemas operativos y sistemas de comunicaciones están escritos en C.

Como comentario previo a la descripción es necesario aclarar que existe confusión sobre si este lenguaje es o no de alto nivel. En este libro se ha dicho que un lenguaje de este tipo está efectivamente por encima de las capacidades expresivas de un lenguaje ensamblador —lo cual definitivamente sí sucede con el lenguaje C— y, en este sentido, no hay confusión posible y C es un lenguaje de alto nivel. Sin embargo, por otro lado, la denominación de la "altura" de un lenguaje también depende de las facilidades que ofrezca para la comunicación con la arquitectura interna de la computadora en la que se ejecute; con sus registros, con localidades de memoria, etc. En este sentido, el lenguaje C ofrece características que lo diferencian de los otros lenguajes de alto nivel. Por ello, a veces se le considera un lenguaje de "nivel intermedio", aunque esto no sea totalmente correcto desde un punto de vista genérico.

C es uno de los lenguajes que más popularidad han adquirido últimamente, y aquí se intentará mostrar algunas de las razones que lo han llevado al lugar que actualmente ocupa.

Antecedentes históricos

C es creación de Dennis M. Ritchie, de los Laboratorios Bell, donde comenzó su diseño a finales de la década de 1960. Los inicios del lenguaje se pueden reconocer en el lenguaje BCPL, creación de Martin Richards, y en el lenguaje B de Ken Thompson, cuyos autores intentaban obtener una

herramienta de programación ágil y moderna. Uno de los objetivos principales en la mente del autor fue disponer de un vehículo de expresión para escribir programación de sistemas y, de hecho, C está íntimamente ligado al sistema operativo Unix, también diseñado por esas fechas en los Laboratorios Bell.

El propio Ritchie lo explica así en un libro que escribiera con Brian Kernighan, y que sigue siendo la referencia estándar sobre este lenguaje (el libro [KERB85], citado en el capítulo 7):

> *C es un lenguaje de programación de empleo general, caracterizado por su concisión y por poseer un moderno flujo de control y estructuras de datos, así como un rico conjunto de operadores. No es un lenguaje de "muy alto nivel" ni "grande", ni está especializado en ningún área particular de operación. Más bien, su carencia de restricciones y su generalidad lo hacen más eficaz y conveniente que otros lenguajes, supuestamente más potentes, para muchas tareas.*
>
> *C fue diseñado originalmente para el sistema operativo Unix en la PDP 11 de DEC e implantado en ella por Dennis Ritchie. Tanto el sistema operativo como el compilador de C y prácticamente todos los programas de aplicación en Unix (incluyendo todo el software con que se preparó este libro) se escribieron en C. [...] Sin embargo, C no está ligado a un hardware o sistema en concreto, siendo fácil escribir programas que ejecuten tareas sin ningún cambio en cualquier máquina que maneje C.*

La realidad es que a este lenguaje le tomó varios años "despegar" y comenzar a ser conocido fuera de su entorno original, cosa que coincidió con la popularización del sistema operativo Unix en las nuevas minicomputadoras.

No obstante que sus autores indican que C no está especializado en ningún área particular de operación, es evidente que los usuarios de este lenguaje sí representan, en términos generales, una sección especializada de los que se dedican a la computación, sección que podríamos llamar de usuarios dedicados al diseño de sistemas científicos o estrictamente computacionales. Aunque es cierto que con cualquier lenguaje es posible escribir cualquier aplicación, no es aún frecuente ver sistemas administrativos escritos en C, y sí es mucho más notoria su utilización en ambientes universitarios o de investigación y desarrollo.

C es un lenguaje de alto nivel, tanto como Pascal, Algol o PL/I, en el sentido de que dispone de todas las características de estructuración, modularidad y legibilidad, entre otras. Es más, prácticamente cualquier aplicación existente en alguno de esos lenguajes puede ser traducida a C con un mínimo de esfuerzo, ya que éste dispone de todos los elementos usuales en un lenguaje de programación (pero excluyendo características especializadas, tales como concurrencia o estructuras de clases al estilo de Ada o Modula-2).

Se presentan a continuación algunas de las características generales del lenguaje, y se pide al lector que las compare con las expuestas en el capítulo 8 para FORTRAN y Pascal.

Estructuras de datos

C se caracteriza por su rico conjunto de operadores que permiten la expresión de algoritmos para manejo detallado de estructuras de datos, apuntadores, listas, etc., junto con todas las estructuras de control de la programación estructurada que se han visto en el texto.

Este es el conjunto de sus estructuras de datos:

Variables sin signo, enteras, de caracteres, reales, dobles

Es de notar que C no maneja cadenas de caracteres en forma directa y que, por tanto, el programador acostumbrado a las facilidades tipo BASIC se sentirá un tanto incómodo al menos hasta que aprenda la manera de lograr los mismos efectos mediante el uso de apuntadores.

Constantes sencillas, dobles, octales y de caracteres

Una constante (o variable) doble es de doble precisión, necesaria para manejos aritméticos especiales.

Las constantes octales son útiles para el manejo de caracteres "raros" que normalmente no pueden ser representados mediante una letra visible (<RETURN> es un caso típico).

Operadores aritméticos, relacionales y lógicos

Se dispone de los cuatro operadores aritméticos usuales, además del módulo y el signo – unario. Los operadores relacionales y lógicos incluyen manejo de bits, y se pueden combinar para lograr una gran cantidad de posibilidades, que se acercan mucho a lo que se puede obtener manipulando bytes directamente en ensamblador.

Conversión entre diferentes tipos de variables

El compilador se encarga de que las mezclas de variables con tipos diferentes queden siempre bien, en el sentido de que no produzcan errores durante la ejecución.

Existe también la posibilidad de forzar temporalmente el tipo de una variable al tipo de otra, pero sin que efectivamente cambie o deje de ser lo que era.

Operadores para asignación de expresiones

En este campo C dispone de un conjunto de operadores especiales de asignación; con ellos es posible hacer manejos y combinaciones fuera

de lo común. Por ejemplo, cualquier variable puede ser incrementada en uno (o decrementada) de manera directa mediante el operador `++` (o bien `--`) que, a su vez, puede ser empleado de manera prefija o sufija, esto es, el incremento se hace antes —o después— de haber usado la variable. Por ejemplo, la expresión

```
if(alfa[i++] == 0) ....
```

compara con cero el valor del i-ésimo elemento del arreglo `alfa` (denotado como `alfa[i]` para sumarle inmediatamente después la unidad al índice `i` y prepararse para el siguiente elemento.
Por otro lado, la expresión

```
if(alfa[++i] == 0) ....
```

compararía con cero el elemento `i+1` de `alfa`, ya que el incremento se efectúa ahora de manera prefija.

Existe también la posibilidad de aplicar las operaciones aritméticas sobre expresiones más complejas de una manera que resulta muy concisa y conveniente.

Expresiones condicionales ternarias

La estructura de selección (si-entonces-otro) puede ser expresada en C mediante un operador ternario (esto es, que requiere de tres operandos), para lograr una escritura parca y concisa (aunque a veces se puede abusar de esta facilidad y volverla en contra del lector). Por ejemplo, este fragmento de Pascal

```
if a > b then z := a
         else z := b ;
```

puede expresarse así en C:

```
z = (a > b) ? a : b ;
```

en donde lo que está entre el signo de interrogación y los dos puntos representa el segundo operando, y lo que viene luego de los dos puntos representa el tercero (el asociado con la cláusula `else`).

Es importante mencionar que se trata de un operador más y que, por tanto, puede ser incluido dentro de expresiones cualesquiera.

Arreglos

En C, la expresión de un vector (o arreglo de una dimensión) se hace de la misma manera que, por ejemplo, en Pascal: definiendo la va-

riable, dotada de un subíndice, como se usó `alfa[i]` en un ejemplo anterior. La dimensión de un vector se fija durante la compilación y el subíndice comienza a partir de 0, no de 1 como en FORTRAN.

Los arreglos de más de una dimensión se expresan de manera un poco diferente de lo usual en otros lenguajes, porque en C existe una liga muy estrecha entre arreglos y apuntadores. Por ejemplo, una matriz de tres renglones por cuatro columnas es en realidad un vector que tiene como elementos tres vectores de cuatro posiciones cada uno, y suponiendo que contiene números reales, se declara entonces así:

```
float alfa [3] [4] ;
```

Obsérvese que los subíndices se escriben por separado, cada uno dentro de su propio par de corchetes.

De hecho, en C, el nombre de un arreglo es en realidad un apuntador a su primer elemento, lo cual permite hacer aritmética de apuntadores y manipulación de arreglos y apuntadores de múltiples formas, que ya no se describen aquí. Baste decir que esta es una de las características importantes de C, ya que permite un considerable ahorro de recursos computacionales, moviendo apuntadores en lugar de contenidos de celdas de memoria. Cualquier operación que se haga mediante subíndices puede lograrse también (y casi siempre con algún tipo de ahorro) usando apuntadores.

Si el programador lo desea, sin embargo, puede trabajar con arreglos de forma equivalente a lo que se hace, por ejemplo, con Pascal.

Manejo de apuntadores

Un apuntador es una variable que contiene la dirección de otra y que, por tanto, puede ser empleada para hacer referencias y manipulaciones indirectas.

Esta es una de las características sobresalientes del lenguaje: de hecho, es difícil *no* usar apuntadores ya que, por ejemplo, cualquier operación de lectura con formato los requiere; igualmente, el paso de parámetros por referencia exige el uso de apuntadores. Es cierto, no obstante, que muchas aplicaciones de los apuntadores son avanzadas, y no aparecen con demasiada frecuencia en la práctica cotidiana de la informática y el procesamiento de datos. Cuando se trata, sin embargo, de aplicaciones científicas, de manejo de esquemas de memoria, de telecomunicaciones o de interfaces con sistemas operativos, entonces sí se vuelve imprescindible emplear esta característica.

Existen dos operadores para emplear apuntadores: el primero (el signo `&`) da la dirección de una variable ya existente, y el segundo (el signo `*`) da el contenido de cierta dirección. Por ejemplo, la declaración

```
int *ap ;
```

define un apuntador (llamado ap) a enteros. Si se desea que haga referencia a cierta variable entera predeclarada (gama), habrá que "cargarlo" con su dirección:

```
ap = &gama
```

ahora ap apunta a gama y puede emplearse para seguirle la pista durante la ejecución. Esto es mucho más útil cuando lo que se está apuntando es un arreglo porque entonces, como se ve más adelante, es posible pasearse a lo largo y ancho del arreglo con facilidad y economía.

Estructuras de datos no homogéneas

Cuando se necesita almacenar bajo un solo nombre un conjunto de elementos heterogéneos, se emplea una estructura, que es similar al record de Pascal o de COBOL.

Esta sería la declaración de una entrada típica en una lista de alumnos, por ejemplo:

```
struct entrada {
        char ap_paterno[20] ;
        char ap_materno[20] ;
        char nombre   [25] ;
        long clave ;
        float calif[10] ;
} ;
```

en la cual se almacena, dentro de un prototipo llamado entrada, el nombre y apellidos, una clave numérica de identificación y hasta diez calificaciones. Luego es posible declarar la existencia de un grupo de cien alumnos:

```
struct entrada alumno[100] ;
```

y hacer referencia a sus campos de manera individual. Así,

```
alumno[k].nombre
```

es una variable de caracteres que contiene el nombre del alumno k-ésimo, y

```
alumno[15].calif[6]
```

contiene la sexta calificación del quinceavo alumno.

Es posible anidar estructuras y también se pueden declarar apuntadores a ellas, e igualmente es factible hacer una serie de combinaciones

que pueden llegar a ser muy complejas y elaboradas, como estructuras recursivas (que hacen referencia a ellas mismas), arreglos de estructuras, campos, uniones y otros manejos avanzados.

Estructuras de control

Por lo pronto se ha hablado tan sólo de la forma en que C permite definir los datos con los que va a trabajar un programa, y ahora se explicará cómo se pueden manipular esos datos ya declarados.

Las estructuras son como las ya explicadas en este texto, y se describen a continuación:

Secuenciación

Una instrucción ejecutable cualquiera de C puede aparecer en cualquier columna del renglón, y termina con un punto y coma. Asimismo, es posible escribir varias proposiciones o instrucciones dentro del mismo renglón, separadas por un punto y coma. Si se desea agrupar varias proposiciones para que se incluyan dentro de una estructura de control, habrá que usar los signos especiales { y }, que desempeñan entonces el mismo papel que `begin` y `end` en Pascal.

Los comentarios pueden ocupar cualquier lugar que ocupe un blanco, y se delimitan entre los pares de signos /* y */.

Selección

Existen dos estructuras de control de selección en C:

`if() - else`
y
`switch - case`

que permiten la expresión completa de algoritmos estructurados, porque además se pueden combinar con cualesquiera de las otras.

El `if()-else` es muy parecido al de los otros lenguajes (Algol, COBOL, PL/I, Pascal), con la particularidad de que no existe la palabra `then`, porque la condición asociada con el `if` debe ir entre paréntesis. A diferencia de Pascal, en C sí se usa el punto y coma antes del `else`, porque la sintaxis pide un punto y coma luego de cada proposición. Así pues, estos dos fragmentos de programa son equivalentes en C y Pascal:

C

```
if( alfa == 88 )
     beta = 6 ;
else {
     beta = 0 ;
     zeta = 2 ;
};
```

Pascal

```
if alfa = 88 then
              beta := 6
             else begin
              beta := 0 ;
              zeta := 2
                end ;
```

Por otro lado, la estructura de control `switch-case` permite hacer selección múltiple, probando el valor de una expresión contra etiquetas constantes para ver cuál alternativa se escoge. A diferencia del `case` de Pascal, el hecho de haber escogido una posibilidad no fuerza a abandonar toda la estructura, por lo que el programador debe terminar cada alternativa con la palabra especial `break`, si es que desea sacar el flujo de control e impedir que continúe probando alternativas.

Así se ve una de estas estructuras, comparada con Pascal:

C

```
switch (digito) {
  case 0: ;
  case 1: alfa = 8 ; break ;
  case 2: DOS ;
  case 3: beta[i++] = 6 ;
          break ;
} ;
```

Pascal

```
case digito of
  0: ;
  1: alfa := 8
  2: DOS ;
  3: begin
     beta[i] := 6 ;
     i := i + 1
     end
end ;
```

Iteración

Se dispone de varias estructuras de control iterativas que se exponen enseguida, comenzando con una muy parecida a la empleada en Pascal y pasando luego a otras un tanto raras.

Éste es un ejemplo de una iteración condicional que obliga al flujo de control a repetir tres instrucciones mientras la condición de control sea verdadera. Está escrito en C y en Pascal:

C

```
while (alfa < beta)
       k = 1 ;
       m = 2 ;
       p = 3 ;
} ;
```

Pascal

```
while alfa < beta do
 begin
  k := 1 ;
  m := 2 ;
  p := 3
 end
```

Sin embargo, la expresión más usada en C, con mucho, se llama for, y tiene esta sintaxis:

```
for( expr1; expr2; expr3 )
    proposición
```

Su traducción a estructuras más sencillas de C es:

```
expr1 ;
while( expr2 ) {
      proposición
      expr3 ;
```

Esto dice lo siguiente: primero, ejecuta la expresión 1; luego, mientras la expresión 2 sea verdadera, ejecuta la proposición y la expresión 3. Como puede verse, aquí se combinan varias cosas, lo que da lugar a una herramienta muy poderosa.

Estos tres fragmentos son iguales, en C, Pascal y FORTRAN:

C

```
for( i = 0; i < 10; i ++ )
   alfa[i] = beta[i] = 0 ;
```

Pascal

```
for i := 1 to 10 do
  begin
    alfa[i] = 0 ;
    beta[i] = 0
  end ;
```

FORTRAN

```
      DO 100 I = 1,10
        ALFA(I) = 0
        BETA(I) = 0
  100 CONTINUE
```

Sin embargo, con el for de C se pueden hacer multitud de cosas diferentes, ya que no está controlado numéricamente, como en FORTRAN o en Pascal, sino que las expresiones 1, 2 y 3 pueden ser cualquier cosa válida. También es posible eliminar alguna de ellas, dando lugar a combinaciones muy eficaces, sobre todo cuando se incluyen llamadas a funciones. Por ejemplo, con esto se calcula la longitud de una cadena de caracteres declarada como `char *s` (apuntador a caracteres) y terminada con un carácter especial llamado nulo y representado por el carácter octal cero:

```
for( n = 0 ; *s != '\0' ; s++ )
     n++ ;
```

Aquí dice lo siguiente: Desde el principio, la longitud es cero; luego, mientras el carácter apuntado por `s` sea diferente del terminador nulo, se incrementa la cuenta en uno y el apuntador avanza al siguiente carácter de la cadena.

Es justo decir que el lenguaje C es muy potente y conciso, pero no siempre es claro a primera vista.

Existen otras estructuras de iteración condicional en C, pero son variaciones sobre lo ya expuesto, por lo que no se abundará en ellas. Sus nombres son `do-while`, que es equivalente al `repeat-until` de Pascal; `break` y `continue`, que se usan para sacar el flujo de control de una iteración, en el primer caso, o para obligar a saltar a la siguiente iteración dentro de un ciclo, en el segundo.

Por último, en C sí existe el `go to` aunque, como ya se ha explicado en este libro, su uso es mínimo, ya que el poder expresivo de las estructuras recién expuestas es suficiente para definir cualquier algoritmo, por complicado que sea.

Modularidad

El manejo de módulos en C es bastante sencillo, digamos, algo intermedio entre el propio de FORTRAN (estructuras estáticas definidas durante la compilación) y el de Pascal (bloques anidados). La estructura global de un programa obliga a que exista un módulo principal (llamado, precisamente, `main`) del cual dependen todos los demás módulos que se vayan a escribir. El programador puede entonces —guiado por algún criterio de diseño— definir tantos procedimientos o funciones como se requieran, estableciendo un sistema de intercomunicación entre las variables que contengan, dependiendo de cómo hayan sido declaradas. Es posible tener declaraciones estáticas, automáticas y globales.

Debido a la estructura de bloques (aunque, como se dijo, no pueden anidarse como en Pascal o Algol), se vuelve necesario definir el alcance de las variables en los diferentes niveles de definición sintáctica, para lo cual se pueden declarar en alguna de las siguientes formas:

- Globales, para obtener un efecto parecido a un "super-COMMON" de FORTRAN, en donde todo mundo conoce todas las variables.
- Automáticas, que es el manejo usual, en donde una rutina no conoce las variables que no fueron declaradas dentro de ella y donde, además, el valor de la variable desaparece cuando el flujo del programa abandona esa rutina.
- Estáticas, en donde el valor de la variable está protegido y no desaparece aun si el flujo de control sale de la rutina que contiene esa variable, lo cual puede ser útil en algunos casos.

Además, C permite el manejo de funciones que devuelven un valor, funciones que devuelven varios valores, con tipos ajustables, apuntadores a fun-

ciones, funciones que devuelven un apuntador, manejo de argumentos variables, llamadas recursivas, etcétera.

Todos los pasos de parámetros sencillos en C son por valor, lo cual significa que no pueden ser alterados por el módulo que los recibe. Si se desea hacer paso de parámetros por referencia, lo que debe pasarse no es el valor del parámetro sino su dirección, para lo cual se emplean los apuntadores, como ya se había mencionado. Cuando en un programa en C se observa que una llamada a un módulo contiene parámetros precedidos por el signo `&`, se sabe que se está enviando la dirección y que, por tanto, se trata de un paso por referencia. Por ejemplo, la llamada

```
tres( alfa, &beta ) ;
```

hace referencia a un módulo llamado `tres`, y pasa un parámetro por valor —`alfa`— y el otro por referencia. Si el módulo `tres` se ve así:

```
tres( alfa, beta )
  int alfa, *beta ;
{
  .
  .
  .
  alfa = 0 ;
  *beta = 0 ;
  .
  .
  .
} ;
```

significa que la asignación `alfa = 0` no tendrá más que un efecto local (dentro de `tres`), pero la otra asignación sí alterará, definitivamente, el valor de `beta`, ya que es hecha a través de la indirección de un apuntador, afectando la localidad de memoria donde reside `beta`, tanto dentro como fuera del módulo en cuestión.

Cualquier módulo de C puede llamarse de forma recursiva, y existen rutinas de biblioteca (o se pueden hacer con relativa facilidad) para asignar porciones de memoria en forma dinámica, es decir, que dependan de las necesidades de un programa a tiempo de ejecución. Como ya se dijo, es un lenguaje que permite la expresión de algoritmos avanzados y de estructuras dinámicas, como listas y árboles.

Por otro lado, pueden declararse ciertas variables críticas como residentes en algún registro del procesador central, para aumentar la eficiencia del proceso de cómputo, aunque esto, formalmente, no tiene nada que ver con la modularidad.

Funciones de entrada y salida

C ofrece un conjunto de funciones (que no son primitivas) para encargarse de las operaciones de lectura y escritura. Se dispone de opciones para escritura con formato (hay 9 posibilidades, entre ellas: decimal, octal, hexadecimal, cadenas, notación científica, etc.), así como para lectura con formato.

Las funciones usuales son `printf` para escribir, y `scanf` para leer, y sus modos de operación son muy amplios y poderosos, aunque su uso es un tanto brusco y es necesario que el programador se acostumbre a ello. Por ejemplo, no existe el equivalente al formato `X` de FORTRAN para "escribir" espacios en blanco, obligando al programador a definir en el formato de salida tantos espacios como requiera.

Por otra parte, existe una multitud de macros y llamadas al sistema operativo que permiten hacer cualquier operación de acceso a discos, archivos, cintas magnéticas, puertos de entrada/salida de la computadora, terminales, etc., dando al interesado la oportunidad de hacer gran cantidad de cosas desde su programa.

Macroprocesador integrado

C tiene un preprocesador integrado que permite hacer sustituciones textuales e inclusión de archivos fuente y módulos, para lograr mayor claridad y transparencia. De hecho, el programador se acostumbra rápidamente a usar el preprocesador para tareas tales como llamadas a macros y bibliotecas de entrada y salida, y para enlazarse con el sistema operativo Unix. Si a alguno no le gustan los signos que C emplea para agrupar proposiciones (las llaves que abren y cierran), puede redefinirlas incluyendo estos renglones al inicio de su programa:

```
define begin {
define end }
```

con ello, el compilador de C sustituirá cualquier aparición de las palabras `begin` y `end` por sus correspondientes llaves. Existen varias opciones para el preprocesamiento, pero ya no se mencionan aquí.

Verificador semántico (`lint`)

En el sistema operativo Unix existe un compilador especializado (llamado `lint`), cuya función no es generar código sino detectar errores sutiles, que normalmente se le escapan al compilador usual de C. Por ejemplo, es capaz de detectar el uso inconsistente de funciones de lectura o escritura, puede advertir al programador sobre posibles problemas a la hora de transportar el programa fuente de una máquina a otra, y logra, en general, atrapar indefiniciones semánticas que quedan fuera del entorno usual en que se mueve la mayoría de los compiladores.

Acceso total a las bibliotecas y funciones del sistema Unix

Para aquellos que trabajan con el sistema operativo Unix, el lenguaje C ofrece todas las facilidades y potencia de operación desde un programa fuente. Es posible tener acceso a los puntos de entrada del sistema operativo desde C, se pueden modificar atributos de puertos, configuración de archivos, manejo de rutinas de entrada/salida, etc., así como manejar buena parte de las características del sistema operativo desde un programa escrito por el usuario. Existe un módulo adicional que permite el manejo de archivos *Index sequential*, por lo que el lenguaje C puede ser empleado también para aplicaciones administrativas y de manejo de grandes cantidades de datos y registros. La ventaja de hacerlo así es la reducción en el tiempo tanto de procesamiento como de manejo del disco magnético.

Referencias sobre el lenguaje C

En el capítulo 7 se hizo la referencia al libro original sobre C, [KERB85]. Aquí se mencionan varios textos adicionales.

Feuer, Alan, *The C Puzzle Book*, Prentice-Hall, New Jersey, 1982.
En este libro se presentan fragmentos de programas en C y se pregunta, a manera de adivinanza, "¿qué hace este código?". Si el lector tiene acceso a alguno de los libros "serios" sobre C y a un compilador, es una manera divertida e interesante de pasar el tiempo.

Gehani, Narain, *Advanced C: Food for the Educated Palate*, Computer Science Press, Maryland, 1985.
Interesante libro escrito por un investigador de los Laboratorios Bell, en donde fueron desarrollados Unix y C. Es un libro para programadores e incluye, además de la descripción del lenguaje C, varios capítulos sobre las interfaces con el sistema operativo Unix. De la misma editorial (y el mismo año) existe otro libro de este autor, titulado *C: an Advanced Introduction*, que es virtualmente idéntico pero, además, contiene un capítulo sobre programación concurrente.

Hancock, Les y Morris Krieger, *The C Primer*, McGraw-Hill, Nueva York, 1986.
Libro introductorio destinado a principiantes. Contiene numerosos ejemplos breves que ilustran características fundamentales del lenguaje.

Kelley, Al e Ira Pohl, *Lenguaje C*, Addison-Wesley Iberoamericana, México, 1987.
Traducción de un buen libro sobre el lenguaje de programación C. Los autores emplean un método de "disecciones" de los programas expuestos,

con fines de aclarar los detalles de uso y las particularidades de cada ejemplo.

Plum, Thomas, *Learning to Program in C*, Plum Hall, New Jersey, 1983.
Escrito para enseñar a programar a principiantes; hace hincapié en la construcción de programas de aplicación que sean portátiles.

Plum, Thomas y Jim Brodie, *Efficient C*, Plum Hall, New Jersey, 1985.
Libro dedicado a mostrar formas de hacer programación eficiente, en términos de tiempo de ejecución y espacio en memoria. Se trata de un texto que supone conocimientos firmes de programación en C y parte de ahí para mostrar técnicas específicas.

Purdum, J., *C Programming Guide*, QUE Corporation, Indiana, 1983.
Texto introductorio, bastante claro, enfocado al uso del lenguaje en microcomputadoras. Tiene un capítulo sobre manejo de archivos en disco y otro sobre los errores que más comúnmente suelen cometer los programadores principiantes. En buen número de ejemplos se muestran programas equivalentes escritos en BASIC.

Schwaderer, David, *C Wizard's Programming Reference*, Wiley Sons, Nueva York, 1985.
Manual muy detallado para programadores de C. Muestra una variedad de técnicas y trucos de programación, además de describir cada una de las instrucciones del lenguaje, junto con comentarios, sugerencias de uso y advertencias prácticas.

Tondo, Clovis y Scott Gimpel, *The C Answer Book*, Prentice-Hall, New Jersey, 1985.
El libro original de Kernighan y Ritchie contiene gran cantidad de ejercicios al final de cada capítulo, y éste contiene todas las respuestas, muchas de las cuales son programas completos. Se trata entonces de un complemento casi indispensable para la referencia original sobre el lenguaje C.
Wagner-Dobler, Friedman, *Lenguaje C*, Addison-Wesley Iberoamericana, México, 1987.
Concisa guía sobre uso y manejo del lenguaje C. Tiene una excelente sección que resume el funcionamiento de muchas de las posibles llamadas a funciones, que pueden llegar a ser verdaderamente confusas para el no experto.

Waite, Mitchell, Stephen Prata y Donald Martin, *C Primer Plus*, Howard W. Sams Co., Indiana, 1984.
Este texto se autodefine como "amigable con el usuario", por lo que se supone que sus ilustraciones y comentarios lo hacen de fácil lectura. Se trata de un buen libro, amplio y sencillo.

Ward, Terry, *Applied Programming Techniques in C*, Scott, Foresman Co., Illinois, 1985.
En este libro no se enseña el lenguaje sino que se describen proyectos interesantes para emplearlo. Es de los pocos libros que describen el diseño, por ejemplo, de un editor de textos o un sistema de intercomunicación entre microcomputadoras.

Glosario mínimo

(Nota: De ninguna manera hay que considerar este pequeño glosario como completo o exhaustivo. Parte de su función es precisamente servir de motivación para averiguar más sobre un tema, ya sea en el texto o en alguna de las múltiples referencias. Se imprimen en **negritas** los términos que a su vez se encuentran descritos aquí, con el fin de establecer referencias cruzadas.)

Algoritmo Desde un punto de vista elemental, un algoritmo no es más que la especificación, detallada y libre de ambigüedad, de un proceso; es decir, un conjunto de pasos que hay que seguir para llegar a cierto fin medible o comprobable.

El problema con esta definición es que la frase "especificación detallada y libre de ambigüedad" es, a su vez, ambigua (¡!), porque en ningún lugar se dice precisamente cómo debe construirse tal especificación, sino que se deja a la interpretación del lector. Para resolver este problema, el matemático inglés Alan Mathison Turing (1912-1954), entre otros, postuló una teoría formal cuyo fin es eliminar la subjetividad de la definición.

Los algoritmos son la base sobre la cual se escriben los **programas**, que una computadora ejecutará para resolver los problemas que se le plantean; en este libro se propone un método general para diseñar algoritmos, por medio de un lenguaje especial llamado **pseudocódigo**.

Analógico, Fenómeno, Modelo Se dice de los procesos que se presentan de forma continua, sin solución de continuidad, a diferencia de los **digitales**, que se muestran en forma cuantizada o discreta. Un modelo analógico emplea funciones matemáticas continuas (diferenciales o integrales) para describir o simular algún elemento de la realidad, que puede ser descompuesto en partes tan pequeñas como se desee, sin perder por ello sus características globales. Para dar un ejemplo simple: una señal eléctrica puede ser usada para transmitir información telefónica analógica, empleando variaciones de voltaje como indicadoras del volumen; si el voltaje sube una centésima, el volumen sube también una centésima, y hay un rango casi ilimitado de posibilidades de variación, que no cambian cualitativamente la señal original.

Archivo *(file)* Conjunto de elementos de información (en forma numérica o alfabética) almacenados en algún medio adecuado, generalmente un disco o una cinta magnéticos. Un archivo típico contiene, por ejemplo, nombres de empleados junto con sus números de registro del seguro social, de tal forma que se puedan escribir programas que utilicen esa información para algún fin particular.

Un archivo está formado de cierto número de esos elementos de información (llamados registros), organizados de determinada forma. Para el caso del archivo de nombres, éstos pueden estar en orden alfabético, o en orden creciente por su registro numérico, etc. La organización de los archivos es un tema importante del quehacer computacional, y existe un gran número de **algoritmos** para crearlos y manipularlos de formas diversas.

Autómata Modelo matemático empleado para ejecutar **algoritmos**. Los autómatas (que no tienen nada que ver con los robots) se agrupan en familias, dependiendo de su complejidad y capacidad de reconocer lenguajes más o menos elaborados. El autómata más genérico y poderoso recibe el nombre de **máquina de Turing.**

Banco de datos, bases de datos (*Data Base Management Systems*) El término *base de datos* puede usarse como sinónimo de banco de datos o banco de información, y se refiere a un conjunto de **archivos** organizados de tal forma que permitan guardar y extraer información útil por medio de la ejecución de programas especiales.

También puede emplearse de forma más técnica y referirse entonces a un sistema de uso general que sirve para crear y mantener bancos de datos sin necesidad de escribir programas específicos para manejarlos, sino usando las facilidades integradas del manejador de la base de datos. Un manejador de una base de datos (DBMS, en inglés) es entonces un sistema complejo que se encarga de interrelacionar los diversos archivos de un banco de información para que éste se comporte como si estuviera dotado de cierta inteligencia que le permite responder preguntas acerca de sus contenidos.

En esta segunda acepción, una base de datos bibliotecaria, por ejemplo, es un sistema al que se le pueden hacer preguntas del tipo "cuántos libros tenemos de psicología infantil", o "cuáles libros tenemos escritos por Jean Paul Sartre", o cosas por el estilo. Como puede comprenderse, el estudio y diseño de las bases de datos es un tema avanzado de las ciencias de la computación, y representa una buena oportunidad para convertir en realidad una de las promesas de la **informática** y acercar efectivamente las computadoras a las necesidades de la sociedad que las emplea, y no tan sólo servir de instrumentos para especialistas.

Binario, Sistema Si los seres humanos tuviéramos tan sólo dos dedos en cada mano, nuestro sistema natural para contar sería binario y no decimal.

Un sistema de numeración binario emplea el 2 como base, y representa todas las cantidades por medio de combinaciones de sólo dos símbolos. Realmente no es tan importante cuáles sean éstos, siempre y cuando sean única-

mente dos diferentes. Para una computadora es mucho más sencillo emplear el sistema binario que cualquier otro porque los dispositivos electrónicos con los que está construida lo permiten de una manera casi natural. La clave Morse que se emplea en la telegrafía es un sistema binario que no sólo sirve para representar números, sino también letras y símbolos, logrando con ello comunicar información con un mínimo de elementos primitivos o básicos (el punto y el guión).

Haciendo una analogía, en una computadora se emplea como punto un impulso eléctrico, y como guión su ausencia. Todo lo demás no es sino variaciones sobre este esquema.

Bit Un bit es precisamente la cantidad mínima de información que un circuito electrónico puede representar, y es la base de operación **binaria** de las computadoras. Esto es, todos los componentes de un sistema computacional están diseñados para almacenar, producir o comunicar bits, que a su vez representan información más compleja. Resulta claro entonces que se requiere un gran número de bits (elementales, básicos, primitivos) para comunicar información simbólica o compleja, porque la capacidad expresiva de un bit es ínfima. De hecho, para representar la letra A se requieren ¡ocho bits!

Booleana, Álgebra Método matemático para tratar y describir las propiedades de un sistema formal de pensamiento que únicamente permite dos posibilidades discursivas: sí y no (o cero y uno). Como el lector podrá comprender, no es mucho lo que se puede argumentar con tan poca materia prima para el discurso, pero es suficiente para la forma en que trabajan los circuitos lógicos de las computadoras. En efecto, toda operación aritmética o lógica (una suma, por ejemplo) puede descomponerse en pasos elementales, aptos para tratarse mediante esta "álgebra de lo sencillo". El creador de todo esto fue el matemático inglés George Boole (1815-1864), en su libro *An Investigation on the Laws of thought on which are founded the Mathematical Theories of Logic and Probabilities*, publicado en 1854, del cual se hizo una semblanza en el capítulo 5.

Boot Este término hace referencia al truco mediante el cual se arranca una computadora para que pueda comenzar a ejecutar programas. Es equivalente, en términos conceptuales a la forma en que un pequeño motor eléctrico arranca al motor de gasolina de un automóvil, y la descripción es demasiado elaborada para incluirla aquí, por lo que se pide al lector consultar la sección 4.4, dedicada a los **cargadores**. Otros nombres que recibe son *bootstrap* e *IPL* (*Initial Program Load*: carga del programa inicial).

Byte Como tan sólo es posible decir dos cosas diferentes por medio de un **bit**, éstos suelen agruparse en unidades más poderosas, llamadas bytes. Un byte está generalmente formado de ocho bits, que permite expresar más posibilidades diferentes. ¿Cuántas cosas se pueden representar con un bit? Sólo dos (0,1). ¿Cuántas con dos bits? Cuatro (00,01,10,11), etc. Con un byte, entonces, es posible decir $2^8 = 256$ cosas diferentes, lo que definitivamente

tiene más utilidad. En general, las computadoras son capaces de manejar bytes de forma unitaria, es decir, procesan ocho bits al mismo tiempo.

Canal En su primera acepción, es el medio físico mediante el cual se transmite información de un punto a otro. El ejemplo más simple de un canal es un alambre que conduce impulsos eléctricos, y una versión mucho más compleja es una red telefónica que, finalmente, sirve para el mismo fin.

En computación, la palabra *canal* también designa a una minicomputadora de propósito especial que se encarga de atender las operaciones de **entrada/salida** y de acceso a los discos magnéticos en un sistema de cómputo grande, liberando al **procesador** central de esta tarea y permitiéndole así dedicarse casi por completo a ejecutar los procesos de los usuarios.

Cargador (*loader*) Programa especial, parte del **sistema operativo**, que tiene como propósito colocar en la **memoria** información codificada en **lenguaje de máquina**, para que entonces la computadora pueda procesarla.

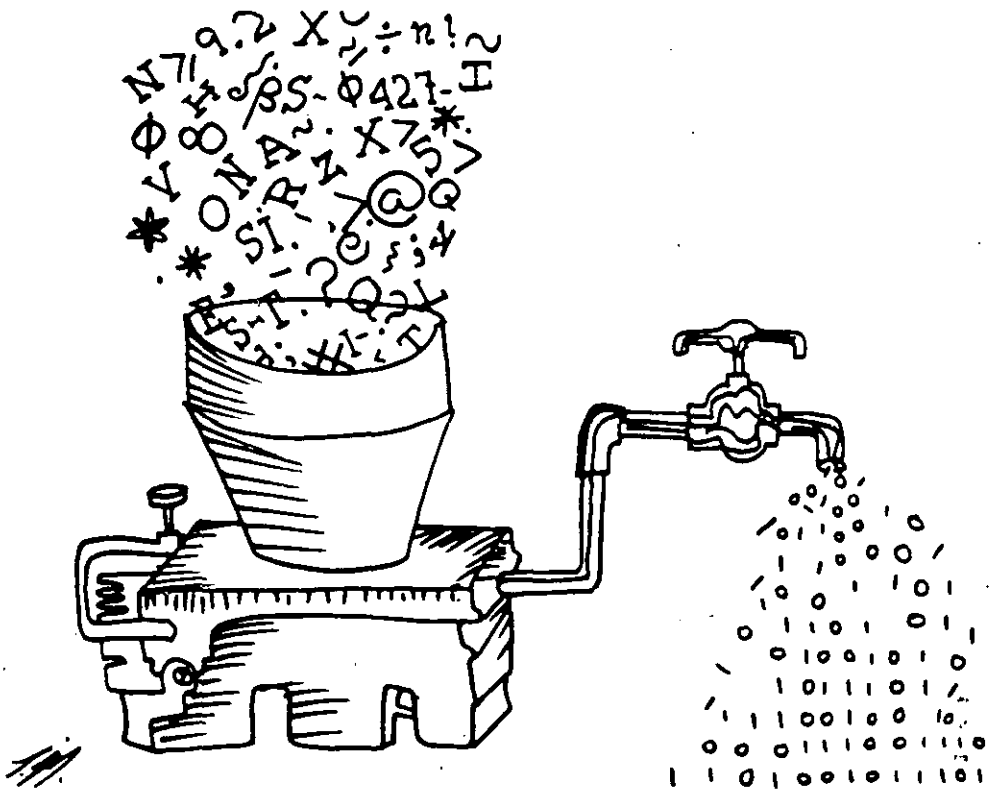
Codificación Con esta palabra se designan varias cosas, en distintos niveles de complejidad. En el nivel más sencillo, codificar es un proceso mediante el cual se traduce información de un lenguaje a otro, usando un diccionario de equivalencias. Tratándose de programación de computadoras en general, la codificación es la última etapa del complejo conjunto de acciones requeridas para resolver un problema por medios computacionales.

Código Se llama así a un programa que está escrito en **lenguaje de máquina**. A los programadores (humanos) les es extremadamente difícil y molesto escribir código, siendo mucho más fácil escribir **programas fuente** en algún **lenguaje de programación**. Será entonces el **ensamblador** o el **compilador** el encargado de generar el código, es decir, de traducir el programa fuente a lenguaje de máquina.

Compilador Programa que sirve como traductor entre un **lenguaje de programación** y el **lenguaje de máquina** de una computadora. Si máquinas diferentes disponen de un compilador del lenguaje A (que traduce **programas fuente** escritos en A al lenguaje de máquina particular de cada una), entonces el mismo programa fuente puede ser compilado y aceptado por computadoras de marcas, tipos y modelos diferentes, lo cual de otro modo sería casi imposible de lograr, dada la enorme cantidad de detalles que varían entre una máquina y otra.

Para una computadora en particular existe un compilador dedicado a cada uno de los lenguajes que pueda manejar. Esto significa que un centro de cómputo donde se procesen programas escritos en seis diferentes lenguajes de programación debe contar con seis diferentes compiladores.

Computabilidad Problema eminentemente teórico, que intenta contestar la siguiente pregunta: "¿Cómo podemos describir —y a qué problemas nos enfrentaremos si lo hacemos— procesos de la realidad que nos rodea, de for-



ma tal que un tercero pueda seguir la descripción con un grado de exactitud que le permita reproducirla?''.

Se trata más bien de un problema filosófico, de la teoría del conocimiento, que puede ser tema de un tratamiento matemático para explorar los límites a los que llegan nuestros mecanismos lógicos y nuestro lenguaje en la interminable tarea de decir algo coherente acerca del mundo.

Simplificando bastante, la teoría de la computabilidad establece que un proceso puede ser descrito mediante un modelo matemático construido con un lenguaje simbólico sencillo, y que existen métodos formales para reproducirlo por medio de un **autómata**.

Más aun, un resultado central de esta teoría descubre la existencia de ciertos procesos que no pueden ser reproducidos ni siquiera por intermedio del autómata más poderoso existente y que, por tanto, están más allá de lo que forma el universo de conceptos cognoscibles en forma algorítmica por el ser humano. Estos problemas reciben el nombre de *indecidibles*. Se ha dedicado el capítulo 5 de este libro a describir algunas de sus características.

Computadora Sistema que, a razón de varios millones de veces por segundo, ejecuta los deseos, caprichos y veleidades de los programadores, así como sus errores y aciertos. Artefacto peligroso por definición.

Desde otro punto de vista, una computadora es la construcción ingenieril del modelo matemático de la computabilidad, y de sus consideraciones filosóficas.

Diagrama de flujo (*flowchart*) Método gráfico para describir **algoritmos**. En principio, todo algoritmo puede representarse mediante un diagrama de flujo, pero en la práctica este acercamiento tiene muchas desventajas, mismas que han conducido a su casi total abandono, en favor de la técnica del **pseudocódigo**.

Digital, Fenómeno, Máquina, Modelo Cuando un fenómeno se comporta de manera discontinua, presentando estados aislados en tiempos bien definidos, se dice que es digital o discreto, a diferencia de aquellos que se presentan o cambian de manera gradual o continuada (**analógicos**). Con mayor precisión, un fenómeno es digital cuando entre cualesquiera dos de sus estados no se puede encontrar un tercero intermedio.

Un modelo digital se usa para describir fenómenos digitales; las máquinas que emplean el sistema **binario** (que reconoce tan sólo dos estados) para trabajar con los fenómenos y modelos digitales reciben, entonces, el nombre de computadoras digitales.

Dirección, Direccionamiento Número de la localidad de **memoria** en la que se almacena un valor. La **unidad central de procesamiento** requiere conocer la dirección de cada celda de memoria que se desee leer (o escribir) durante la ejecución de un programa, y a las diferentes formas de lograr esto se les conoce como métodos de direccionamiento.

Diseño Estructurado Conjunto de métodos para construir **sistemas de información** o, en general, para resolver problemas por medio de una computadora. El diseño estructurado parte de la base de que los programas se escriben estructuradamente, pero no se limita a consideraciones locales al programa, sino que estudia las relaciones entre todos los programas que constituyen un sistema.

Editor Programa de uso general por medio del cual se crean textos o programas que le son alimentados a una computadora para su posterior procesamiento. Existen dos tipos de editores: de línea y de pantalla; en los del primer tipo se manipula sólo un renglón (o unos cuantos) del texto que está siendo creado, mientras que en los del segundo se trabaja con una hoja de texto virtual que recoge todos los cambios que se deseen hacer, sin necesidad de especificarlos uno por uno. Un editor de pantalla es mucho más cómodo de emplear, en términos generales, que uno de línea.

Cuando el editor tiene facilidades para definir formatos de impresión del texto (numeración automática de páginas, texto alineado a la derecha, notas al pie, etc.) recibe el nombre de procesador de palabras.

Ensamblador En su primera acepción, un ensamblador es un programa que recibe programas escritos en un lenguaje (que también se llama ensamblador) y los traduce a **lenguaje de máquina**.

La segunda acepción se refiere precisamente al lenguaje que reconoce un programa ensamblador, y que constituye el segundo nivel (en complejidad creciente) de los lenguajes con que se puede programar una computadora.

En inglés se puede diferenciar entre las dos acepciones, porque la primera se conoce como *assembler* y la segunda como *assembly language*, aunque también suelen confundirse ambos términos y emplearse simplemente *assembler*; entonces, el contexto indica cuándo se habla del programa que traduce y cuándo del lenguaje en el que se escribe un programa.

Programar en lenguaje ensamblador (también llamado de bajo nivel) tiene muchas desventajas, porque no permite libertades al programador, que se ve limitado a usar construcciones con muy poco poder expresivo. Otra desventaja del lenguaje ensamblador es que no es portátil, lo que significa que hay un lenguaje ensamblador para cada marca (o modelo) de computadora, y que todos ellos son diferentes e incompatibles.

Entrada/Salida, Dispositivos, Operaciones, Procesador de Cualquier cosa que no forme parte de la **unidad central de procesamiento** o de la **memoria central** de una computadora está fuera del sistema. Por tanto, se requieren mecanismos especiales para tener acceso a la información que entra del exterior o para emitir la de salida. Las operaciones de E/S o I/O (*Input/Output*) se encargan de esto. En las computadoras grandes o complejas existen procesadores especiales que ejecutan estas operaciones, y a veces se les conoce como **canal**.

Los dispositivos de entrada más comunes suelen ser las **terminales** de video y las lectoras de tarjetas (ya casi en desuso), y los de salida más comunes son las terminales de video (que sirven tanto para entrada como para salida), las impresoras y los graficadores.

Estructuras de control Construcciones mediante las cuales se escriben los programas. Cuando las estructuras de control se definen y emplean de manera ordenada y sistemática, se habla de **programación estructurada.** No todos los **lenguajes de programación** disponen de estructuras de control completas o complejas, por lo que la tendencia es a abandonar los que son deficientes en este punto.

Estructuras de datos Métodos que se emplean en programación para organizar y representar la información en una computadora. En términos generales, un programa está formado por el **algoritmo** (que dice qué se va a hacer y en qué orden) y las estructuras de datos (que dicen cómo se manipulan los datos con los que se va a trabajar). Todos los **lenguajes de programación** disponen al menos de un tipo básico o primitivo de estructura de datos (la representación de un número entero, por ejemplo), y cuanto mayor sea el número de estructuras diferentes de datos que ofrezca un lenguaje (arreglos, listas, etc.), tanto más fácil y versátil será la tarea de escribir algoritmos con él.

Fragmentación Problema que se presenta en la **memoria central** de una computadora cuando se asigna un área fija para cada **proceso.** Como no se puede averiguar de antemano el espacio de memoria requerido por los procesos de los usuarios, si el **sistema operativo** emplea un esquema poco ágil

para el manejo de la memoria, entonces se presentará el caso de que haya secciones desperdiciadas, porque le sobran a un proceso pequeño, pero no alcanzan para completar los requerimientos de otro más grande. Los sistemas operativos solucionan el problema de la fragmentación por medio de métodos más complejos de manejo dinámico de memoria, como paginación o segmentación.

Gramática formal Especificación matemática para el generador de un **lenguaje de programación**. El objetivo es poder construir un reconocedor automático de las frases producidas por una gramática, para establecer comunicación entre generador y reconocedor de un lenguaje. El programador aprende las reglas gramaticales del lenguaje de programación y las usa para escribir sus **algoritmos** y programas, que después serán "entendidos" por el reconocedor (**compilador** o **ensamblador**), que las traducirá a **lenguaje de máquina** para su posterior **carga** y ejecución. Las gramáticas de los lenguajes de programación se conocen matemáticamente como independientes del contexto, y en general son objeto de estudios matemáticos de nivel avanzado. En el capítulo 5 se da una introducción al tema.

Hardware Conjunto de elementos y sistemas electrónicos que forman un sistema de cómputo. Al inicio de la corta historia de la computación digital era fácil distinguir el hardware del **software**, aunque a medida que avanzan los desarrollos tecnológicos se vuelve más sutil la barrera que separa uno del otro, porque ahora buena parte de la **programación de sistemas** de muchas computadoras reside en un nivel intermedio (**microprogramado**), que también se conoce como *firmware*.

Informática Conjunto de conocimientos, métodos y sistemas para el manejo "computarizado" de la información en las organizaciones. Se ha hecho un uso abusivo y ligero de este término, confundiéndolo con campos que sólo lo influyen de manera marginal. Aunque ciertamente la legislación del trabajo, por ejemplo, es un aspecto digno de tomarse en cuenta a la hora de pensar y planear una institución pública o federal, tiene poco que ver con el conjunto de conocimientos que deben tener quienes diseñen los sistemas, los flujos de información y los programas de computadora que emplee la oficina para su funcionamiento.

El problema no es que los especialistas en informática estudien métodos de administración moderna o de manejo financiero, sino cuando esto impide, por cuestiones de diseño de programas educativos, dedicar la suficiente atención a las bases formales de la ciencia de las computadoras y el manejo de información.

Tal vez parte del problema estribe en la holgura con la que se emplea este término, donde unos entienden administración moderna, otros derecho a la información y otros sistemas de información por medio de computadoras.

Intérprete Un intérprete traduce, en forma interactiva o dinámica, programas escritos en un **lenguaje de alto nivel** a **lenguaje de máquina** ejecu-

table. La diferencia entre un intérprete y un **compilador** reside en que el segundo traduce el **programa fuente** una sola vez, generando **código** para ser **cargado** y ejecutado posteriormente, mientras que el primero traduce y ejecuta inmediatamente cada uno de los renglones del programa fuente, sin generar código.

Interrupción Mecanismo mediante el cual el **procesador** abandona la tarea que estaba ejecutando y se dedica a atender otro **proceso** cargado en la memoria. De hecho, las interrupciones son la base sobre la que se diseñan los **sistemas operativos** de **multiprogramación**, y forman parte central de su estudio. Durante un segundo de la operación de un sistema de cómputo complejo, el procesador central puede ser interrumpido varias decenas o centenas de veces, y esto no representa ninguna anomalía, sino que es la forma normal de trabajo.

Lenguaje de alto : Se llama así a los **lenguajes de programación** que están "por encima" (en poder expresivo) del **lenguaje de máquina** y del lenguaje **ensamblador**. Existe una gran cantidad y diversidad de estos lenguajes, pero para cada uno de ellos debe existir un **compilador** que traduzca el programa escrito en él al lenguaje de la máquina donde se intente ejecutarlo.

Lenguaje de máquina La única manera de comunicarse con el **procesador** de una computadora es por medio de un programa directamente ejecutable, mismo que debe estar escrito forzosamente en este lenguaje. Sin embargo, el lenguaje binario de máquina no es propiamente un lenguaje, porque carece de estructura; podría describirse más apropiadamente como "conjunto de signos aislados ejecutables". Esto significa que la máquina en ningún momento "sabe" lo que está haciendo, ni si va en el camino correcto para la solución de un problema. Precisamente debido a esta falta de contenido semántico es que ni la programación ni la computación hubieran avanzado de no ser por la 'humanización" que introducen los lenguajes de programación de alto nivel, que hacen realidad el dictado del filósofo Ludwig Wittgenstein: "los límites de mi lenguaje significan los límites de mi mundo" (*Tractatus Logico-Philosophicus*, 5.6).

Lenguaje de programación Nombre genérico que se aplica a cualquier lenguaje (fuera del de máquina) disponible para escribir programas para una computadora. Si no se incluye el **ensamblador**, entonces este concepto es idéntico al de lenguaje de alto nivel. De forma aparentemente paradójica, escribir un programa en algún lenguaje de programación es la última etapa del proceso conocido comúnmente como "programar". Los pasos anteriores son —en orden— entender el problema, analizarlo y programarlo en **pseudocódigo**.

Macroexpresión, macroprocesador Construcción escrita en lenguaje **ensamblador** que se emplea como abreviatura en la programación. Una macroexpresión requiere ser expandida por el macroprocesador para que forme

parte del **programa objeto**. Sugerimos al lector que consulte la sección 4.3 para una explicación, pues ello rebasa los límites de este glosario.

Máquina de Turing Modelo matemático (no es una máquina) de un **autómata** general. Alan M. Turing lo diseñó como parte de sus avanzados estudios sobre **computabilidad** y **algoritmos**, y se emplea como medio de análisis en el trabajo teórico, y como herramienta para probar teoremas y proposiciones en matemáticas computacionales y teoría de lenguajes.

Memoria central Es parte, junto con el **procesador**, de la **unidad central de procesamiento** de una computadora (aunque este nombre a veces designa tan sólo al procesador). Todo programa que vaya a ser ejecutado forzosamente debe residir en la memoria central, en forma ya de **código objeto**, siendo función del **cargador** llevarlo allí. La cantidad de memoria de que dispone una máquina se mide en **bytes**, y los tamaños típicos varían entre varias decenas de miles, para las computadoras pequeñas, hasta cinco o diez millones de bytes, para las grandes instalaciones.

Memoria secundaria Como la memoria central de una computadora es costosa y relativamente escasa, se vuelve necesario disponer de algún medio masivo de almacenamiento, capaz de guardar millones (o cientos de millones) de **bytes** de información a un costo razonable. Esta es la función de la llamada memoria secundaria, o periférica, que está representada por los discos y las cintas magnéticas. En un disco magnético se graban (y reproducen) **bits** con una tecnología similar a la empleada para las grabadoras de audio; la diferencia principal es que aquí se guarda información **digital**, en lugar de **analógica** o común.

Toda computadora requiere algún dispositivo de memoria secundaria para funcionar ya que, a diferencia de la **memoria central**, la información almacenada en los discos sí se mantiene aun cuando la máquina esté apagada. En otras palabras, la memoria secundaria no es volátil, por lo que permite recuperar la información o los programas previamente almacenados. De hecho, todos los programas que forman el **sistema operativo** de una computadora residen originalmente en disco, y se cargan en la memoria central cuando son requeridos. Esta idea da lugar al concepto de **memoria virtual**.

Memoria virtual Si se logra conectar la **memoria secundaria** con la **memoria central**, de tal forma que se establezca un flujo ininterrumpido de información entre ambos mecanismos de almacenamiento (uno rápido y volátil —central—, y otro relativamente lento pero permanente —secundario—), entonces se puede lograr la simulación de una memoria central que sea virtualmente tan grande como el espacio en disco. En otras palabras, los discos alimentan a la memoria principal con la información que el **procesador** requiere para ejecutar los procesos, "engañándolo" para producir la ilusión de que se dispone de un espacio de datos enorme, aunque en realidad lo que sucede es que el sistema operativo se encarga de traer y llevar pequeñas sec-

ciones de información entre ambas memorias, justo un momento antes de que el procesador la requiera.

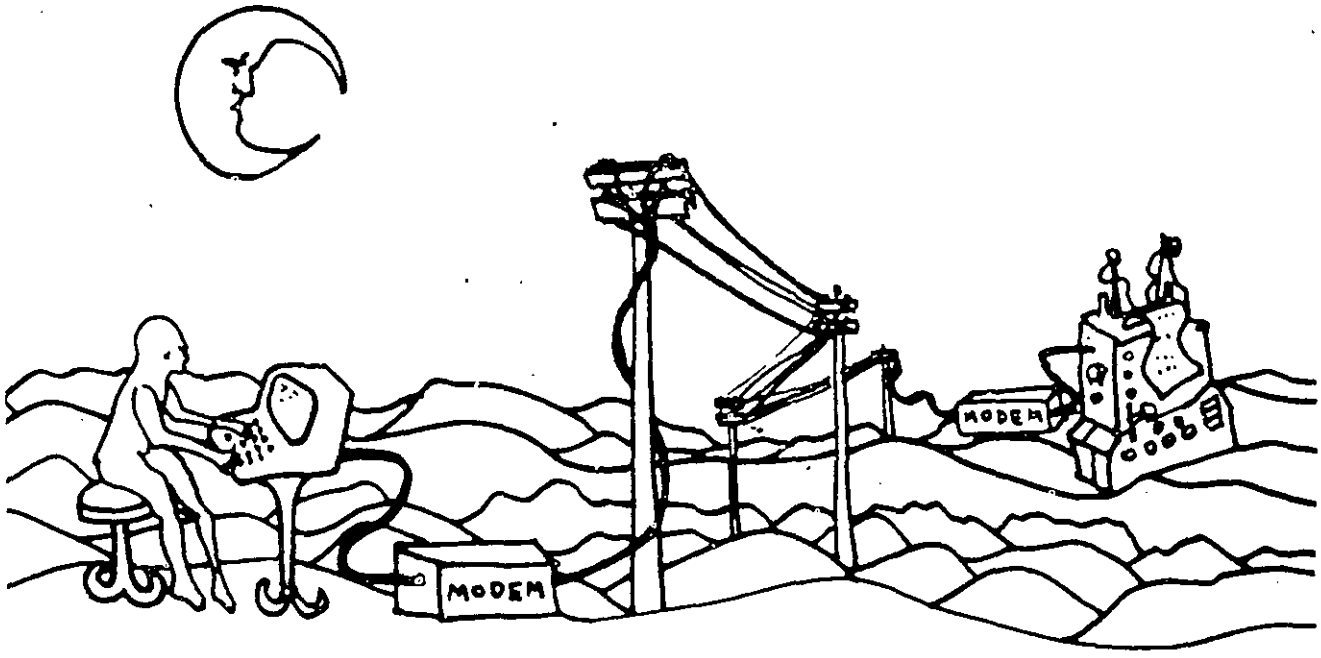
Los mecanismos más usuales de memoria virtual son paginación y segmentación, pero tienen la suficiente complejidad como para que sólo las computadoras grandes y complejas los empleen. Últimamente han comenzado a aparecer **microprocesadores** que efectúan las operaciones básicas de la memoria virtual de forma integrada, (esto es, mediante el hardware) por lo que se espera que en el futuro las máquinas pequeñas y medianas también empleen el mecanismo de memoria virtual, con todas sus ventajas asociadas, como mayor flexibilidad de procesamiento y mejores capacidades de manejo de información.

Microcomputadora Sistema de cómputo consistente en un **microprocesador, memoria central, memoria secundaria** y una **terminal** de video o un televisor, todo en un mismo paquete, de tamaño y costo reducidos, que permiten su empleo en sectores de la sociedad que antes eran casi por completo ajenos al manejo de las computadoras, tales como pequeñas empresas, escuelas y hasta particulares con afanes de entrar en este nuevo mundo.

Microprocesador Circuito electrónico integrado (*chip*) que efectúa las funciones de un **procesador central** completo, con una velocidad inferior a la que funciona una gran computadora, pero con un costo incomparablemente menor. Un microprocesador típico es un circuito que cabe en la palma de la mano, que contiene el equivalente a varias decenas de miles de transistores integrados, que ejecuta operaciones a razón de un millón por segundo, y que cuesta menos de veinte o treinta dólares. No debe confundirse, sin embargo, el microprocesador con la **microcomputadora**, que lo usa como corazón, junto con una buena cantidad de dispositivos adicionales, que incrementan considerablemente el costo total (una microcomputadora de las llamadas "personales" cuesta entre varios cientos y algunos miles de dólares).

Microprogramación Técnica usada para dirigir los pasos que un **procesador** tiene que dar para poder ejecutar una instrucción de **lenguaje de máquina**. Estos micropasos están almacenados en un dispositivo electrónico que forma parte del procesador central, y constituyen lo que se llama el microprograma. Si un procesador está microprogramado (no todos emplean esta técnica), entonces puede, con cambios en su microprogramación, emular a otros. Las computadoras grandes casi siempre emplean esta técnica, que ahorra muchos esfuerzos de ingeniería, pues permite usar instrucciones de un procesador anterior en el diseño de uno nuevo.

Modem Un MODulador-dEModulador es un dispositivo electrónico utilizado para comunicar una computadora con una **terminal** remota (o con otra computadora), empleando líneas telefónicas para establecer el contacto. Un modem se encarga de convertir (modular) la información **digital** que sale de la computadora en información **analógica**, para hacerla compatible con los requerimientos del sistema normal de telefonía, mientras que, en el destino,



otro modem se encarga de traducir de nuevo (demodular) la información analógica que llegó a la forma digital que espera el dispositivo que la recibe.

Multiprogramación Esquema empleado en computadoras medianas y grandes para atender concurrentemente a más de un usuario. Un **sistema operativo** de multiprogramación mantiene **cargados** varios **procesos** en la memoria central, y reparte las capacidades del procesador entre ellos para dar atención a todos en una unidad de tiempo, que casi siempre es un segundo o menos. Esto significa que en un segundo el procesador central ejecuta un fragmento de cada proceso residente en la memoria, dando a los usuarios la impresión de que están siendo atendidos en forma individual. Hay una distinción técnica entre multiprogramación y **tiempo compartido**, que no es relevante desde el punto de vista del usuario, y que consiste en que el sistema de multiprogramación abandona el proceso que pidió una operación (lenta) de **entrada/salida**, para comenzar (o volver a iniciar) de inmediato algún otro proceso. Cuando el procesador detecta la aparición de una de estas operaciones —por medio de una **interrupción**— dirige el pedido al **canal** para que éste lo atienda. Cuando se completa la operación, el procesador es interrumpido de nuevo para que esté en posibilidades de atender de nuevo el proceso en cuestión.

Paquetes de programación Con este nombre se conocen diversos sistemas disponibles en el mercado de computadoras, diseñados para resolver problemas específicos que aparecen normalmente en las organizaciones, tales como cálculo y pago de nóminas, control de inventarios, de clientes, etc. Existen

también paquetes de graficación, de programación matemática, de simulación, de estadística, y casi de cualquier actividad que pueda ser de interés para alguna organización o empresa que dispone de una computadora pero que no puede o no desea invertir tiempo ni recursos de programación para desarrollar esas aplicaciones.

Periféricos Nombre genérico para referirse a los dispositivos de **entrada/salida** de una computadora, así como a sus unidades de **memoria secundaria**. En el caso de las computadoras personales y las **microcomputadoras**, los periféricos casi siempre rebasan en costo al procesador y la memoria central.

Procesador El procesador es el encargado de ejecutar las instrucciones de un programa, escritas en **lenguaje de máquina**. Es un dispositivo electrónico diseñado para efectuar cientos de miles (o millones) de operaciones en un segundo. El procesador central es el corazón de la computadora y controla todas las operaciones que ésta realiza.

Podría decirse que todos los esfuerzos de los programadores y analistas están orientados a tratar de controlar y utilizar el desafío que representa el tener un robot capaz de efectuar un millón o más de órdenes por segundo.

Procesamiento distribuido Con el advenimiento de los **microprocesadores**, al comienzo de la década de 1970, surgió la idea de repartir las tareas computacionales entre varios subcentros, dotándolos de inteligencia y poder de ejecución local. En un sistema distribuido hay varios procesadores, que atienden partes del todo, y que son controlados y coordinados por un **sistema operativo** especial. Cuando algunas partes del proceso tienen lugar en localidades separadas (por ejemplo en un sistema de vigilancia y control sismográfico), entonces se habla de una **red.**

Proceso Conjunto de instrucciones escritas en **lenguaje de máquina** que ya están listas para ser ejecutadas por el **procesador**. Un proceso es el resultado final del complejo conjunto de acciones que comienzan cuando un programador pone manos a la obra y escribe un programa, que luego será traducido por un **compilador**, atendido por el **sistema operativo** y finalmente cargado en la **memoria central**.

Programa fuente (*Source Program*) Conjunto de instrucciones escritas en algún lenguaje de programación y que supuestamente sirven para modelar o solucionar un problema. Decimos "supuestamente" porque en realidad los programas fuente escritos por seres humanos casi nunca consideran todos los detalles, y entonces el problema se resuelve con una iteración de este esquema: programa fuente → compilación → corrección de errores → segunda versión del programa fuente → compilación → corrección de errores → tercera versión, etcétera.

La **programación estructurada** es un intento formal de reducir esta infortunada cadena de sucesos.

Programa objeto Conjunto de instrucciones escritas en **lenguaje de máquina**. Los humanos ya no escribimos programas objeto, porque delegamos esta tarea a los programas traductores: **ensambladores** y **compiladores**. Para que un programa objeto pueda ser ejecutado debe llevarse del disco, donde generalmente está almacenado, a la memoria central.

Programación de sistemas Nombre genérico con el que se conoce el enorme conjunto de programas y sistemas que se encargan de la operación automática de las computadoras. Los componentes más importantes de la programación de sistemas son los **ensambladores**, los **compiladores** y los **sistemas operativos**. El objetivo al que aspira la programación de sistemas es convertir a la computadora en un compañero natural del hombre, cosa que está aún muy lejos de suceder, y que tal vez (y no sabemos si esto sea una bendición o no) nunca suceda.

Programación estructurada Conjunto de métodos para diseñar y escribir programas empleando el método científico y no tan sólo el método de ensayo y error.

La programación estructurada dicta componentes primitivos y reglas de composición que permiten construir programas que cumplan su objetivo a la vez que sean legibles y fáciles de entender y modificar.

Pseudocódigo Mezcla de **lenguaje de programación** y español (o inglés o cualquier otro idioma) que se emplea, dentro de la **programación estructurada**, para realizar el diseño de un programa. La idea del pseudocódigo consiste en aprovechar la flexibilidad y poder expresivo del lenguaje natural, por un lado, y el poder estructurante de las construcciones formales (o reglas de composición), por el otro, para avanzar sobre seguro en la tarea de escribir programas para computadora.

Recursividad Método matemático para definir funciones que consiste en partir de una base e ir construyendo los componentes de la función haciendo referencia a la definición de la función misma, en una especie de "círculo vicioso controlado"[1]. La definición recursiva de una operación como elevar un número a una potencia, por ejemplo, dice así: "Un número elevado a la potencia cero produce la unidad; la potencia de un número se obtiene multiplicándolo por sí mismo elevado a la potencia menos uno". O sea que si se quiere aplicar la definición y elevar 3 al cuadrado, hay que decir

$$3^2 = 3 \times (3^1) = 3 \times ((3 \times (3^0)) = 3 \times (3 \times 1) = 3 \times 3 = 9 ,$$

que puede parecer innecesariamente complicado, pero que, sin embargo, es una definición totalmente elegante, en el sentido de que emplea un mínimo absoluto de elementos externos a la operación que se está definiendo.

Si el lector dice "tres al cuadrado es tres multiplicado por sí mismo", no está dando una definición genérica de la operación, porque para definir "tres a la quinta" tiene que repetir —redundantemente y, por ende, con

poca elegancia— demasiadas veces la frase "multiplicado por sí mismo" para obtener: "tres a la quinta es tres multiplicado por sí mismo, multiplicado por sí mismo, multiplicado por sí mismo, multiplicado por sí mismo". O bien decir cuántas veces hay que multiplicarlo por sí mismo, lo que también quita generalidad a la definición.

La recursividad es un concepto central en matemáticas, y está en la base misma de su fundamentación formal. En el capítulo 5 se citó la referencia [DOUA70], que es una fuente muy accesible y amena donde se tratan algunos de estos temas.

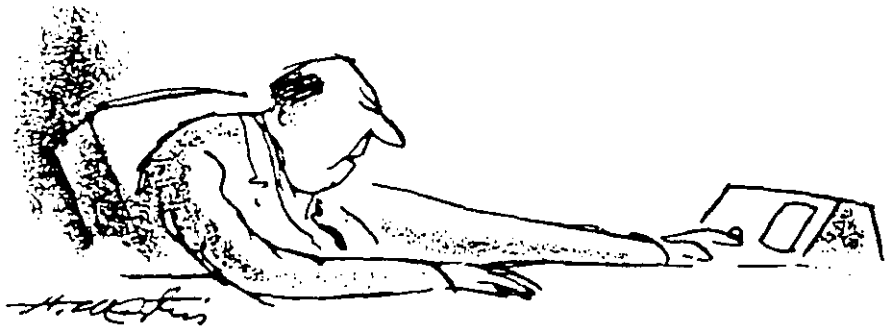
Por otro lado, un **lenguaje de programación** recursivo es un lenguaje que permite definir funciones o módulos de forma recursiva, empleando un razonamiento similar a lo descrito. Técnicamente, se dice que una rutina es recursiva cuando se llama a sí misma durante la ejecución.

Red de computadoras Sistema complejo de **procesamiento distribuido**, configurado con una arquitectura que permite comunicación y transferencia de **archivos** entre las computadoras que lo forman.

En una red es posible interconectar varios sistemas de cómputo completos, de forma tal que un usuario de alguno de ellos pueda tener acceso a la información que maneja cualquiera de los demás.

Relocalización Mecanismo que permite mover un **programa objeto** a diversas posiciones de la **memoria central** sin que se vea afectada su capacidad de ejecución. Recuérdese que, por definición, un programa escrito en **lenguaje de máquina** depende de **direcciones** absolutas y que, si no se convierte en relocalizable, solamente será posible ejecutarlo a partir de la dirección inicial de carga, limitando mucho la flexibilidad del sistema de cómputo. Para que pueda existir un **sistema operativo** de **multiprogramación** es necesario resolver el problema de la relocalización, cosa que se logra por medio de **cargadores** complejos.

Sistema de información Complejo de elementos que interactúan entre sí para manipular, crear y consultar información proveniente de un **banco de datos**. Para diseñar un sistema de información se requieren conocimientos de computación y de **informática**, y de la participación de las áreas demandantes de la información. Por ejemplo, si se deseara tener un sistema de este tipo en un banco, sería necesario trabajar con los encargados de la planeación y administración, para entender con claridad los mecanismos, métodos de organización y funcionamiento de las diversas áreas con las que la institución cuenta. Una vez determinado esto se procedería a interrogar a las áreas usuarias (cajeros, contadores, financieros, subgerentes, etc.) para definir prioridades de atención y necesidades de información. Todo este trabajo aún no tiene mucho que ver con la computadora, sino que más bien se orienta hacia la institución misma. Sólo después de este análisis vendrá la etapa del diseño del sistema computacional que explotará la información bruta (movimientos, depósitos, retiros, contratos, etc.) para obtener informes de utilización, proyecciones financieras, estadísticas y demás elementos que sirvan para la me-



"Señorita Dennison, mándeme algunos informes, gráficos, programas, archivos, datos, índices, hechos, cifras, totales, informes actuariales, análisis de sistemas, proyecciones, folletos, descomposiciones, resúmenes, tasas, registros, estados, memorandos, diagramas, peticiones..." © DATAMATION

jor operación o para la toma de decisiones. En términos generales, el diseño de un sistema de información completo es tarea que requiere meses de trabajo.

Sistema operativo Conjunto de programas que controlan la operación automática de un sistema de cómputo, con el fin de optimizar su funcionamiento y presentar una imagen monolítica y virtual ante sus usuarios. Un sistema operativo transforma una computadora electrónica en un agente procesador de información con un nivel de significancia y coherencia que es más que la suma de las partes que la componen.

El estudio de los sistemas operativos es parte central de la **programación de sistemas**, y representa la oportunidad de acercar la computadora a los usuarios, permitiéndoles manejarla y aprovecharla en formas más versátiles y eficientes.

Software Nombre genérico que se da a los programas de una computadora, pero que implica una responsabilidad adicional: asegurar que el programa o sistema cumple por completo con sus objetivos, opera con eficiencia, está adecuadamente documentado y es sencillo de operar.

Desde este punto de vista, no cualquier programa de computadora califica para obtener este título.

SPOOL, -er, -ing Acrónimo en inglés de *Simultaneous Peripheral Operations On Line* (operación simultánea de periféricos en línea), que designa un esquema que emplean casi todas las computadoras, excepto las más simples, para manejar las operaciones de **entrada/salida** de forma que no desperdicien tiempo del **procesador central**. La idea consiste en interceptar los pedidos de entrada/salida y dirigirlos al disco magnético, evitando así la pérdida de las valiosas fracciones de segundo que tarda una lectora o una impresora en efectuar una operación de lectura o escritura. Como ventaja adicional (que

resultaría muy preciada en las computadoras personales) se eliminan los tiempos de espera por impresión, que a veces son del orden de varios minutos, en los que, cuando no hay SPOOLing, la terminal de la máquina es incapaz de mandar o recibir mensajes al procesador central.

Teleinformática, Teleproceso Técnicas para comunicar computadoras o **terminales** remotas entre sí. El diseño de una **red** de teleproceso debe tomar en cuenta los flujos y cantidades de información, la velocidad de los **canales** de comunicación, la localización de los equipos, y las capacidades del **software** de comunicaciones disponible. Los grandes sistemas de teleproceso emplean métodos de comunicación por microondas y satélites, y se cuentan entre los ejemplos más complejos y sofisticados de tecnología.

Terminal Nombre genérico con el que se conocen los dispositivos de comunicación usados por la mayoría de las computadoras para entablar conversación con los usuarios. Las terminales de video (o pantallas) sirven de medio de **entrada/salida** en un sistema de cómputo, funciones que también cumplen los teletipos, las impresoras (de salida únicamente) y los graficadores, entre otros.

Unidad Central de Procesamiento (CPU) Nombre asignado al **procesador** central de una micro o minicomputadora, pero que también se da al conjunto que forman el procesador y la **memoria central** de una máquina grande (*mainframe*). La distinción entre ambas connotaciones la da el contexto en que se emplean. En todo caso, se refiere a la parte más importante de una computadora.

NOTA FINAL

El proceso de enseñanza-aprendizaje es eso precisamente, un proceso: como en toda tarea compleja y cambiante, aquí hay mucho campo para modificaciones, añadidos y sugerencias, por lo que se aprovecha esta nota de despedida para hacer al lector, estudiante o profesor, una atenta e interesada invitación para que haga llegar al autor en forma personal sus comentarios, quejas o anotaciones sobre el texto, en el entendido de que serán tomadas en cuenta de manera individual.

Favor de escribir a:

Guillermo Levine
Micrológica
Hidalgo 61
San Jerónimo
México 10200, D.F.
México

Resumen de la bibliografía

Para facilitar la consulta de la bibliografía se muestran a continuación, en una sola sección, los títulos de los 109 libros y artículos citados en el texto. De cada uno se hizo un breve comentario al final del capítulo donde aparecen.

Antecede a cada título el número del capítulo donde se cita y se comenta.

Al final se relacionan también otros 47 libros y artículos que se mencionaron en el texto, aunque no como referencias directas.

4 [AHOA77] Aho, Alfred y Jeffrey Ullman, *Principles of Compiler Design*, Addison-Wesley, Massachusetts, 1977.

4 [AHOA85] Aho, Alfred, Ravi Sethi y Jeffrey Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Massachusetts, 1985.

3 [ATKT79] Atkins, Travis, "What is an interrupt?", en *Byte*, marzo, 1979.

4 [BACM86] *Bach*, Maurice, *The Design of the UNIX Operating System*, Prentice-Hall, New Jersey, 1986.

6 [BARN86] Baron, Naomi, *Computer Languages: A Guide for the Perplexed*, Anchor Press/Doubleday, Nueva York, 1986.

4 [BECL88] Beck, Leland, *software de sistemas: Introducción a la programación de sistemas*, Addison-Wesley Iberoamericana, México, 1988.

4 [BENA82] Ben-Ari, M., *Principles of Concurrent Programming*, Prentice Hall International, Londres, 1982.

4 [BERH80] Berliner, Hans, "Computer Backgammon", en *Scientific American*, junio, 1980.

4 [BEVT85] Thompson, Beverly y William Thompson, "Inside an Expert System", en *Byte*, abril, 1985.

2 [BHAR87] Barath Ramachandran, "Information Theory", en Byte, diciembre 1987.

3 [BIGC83] Bigelow, Charles y Donald Day, "Digital Typography", en *Scientific American*, agosto, 1983.

5 [BOUN72] Bourbaki, Nicolás, *Elementos de historia de las matemáticas*, Alianza Editorial, Col. Alianza Universidad, núm. 18, Madrid, 1972.

4 [BRIH73] Brinch Hansen, Per, *Operating Systems Principles*, Prentice Hall, New Jersey, 1973.

7 [BROF75] Brooks, Frederick, Jr., *The Mythical Man-Month*, Addison Wesley, Massachusetts, 1975.

4 [BROP75] Brown, P. J., *Macro Processors*, Wiley & Sons, Londres, 1975.

5 [CHON57] Chomsky, Noam, *Syntactic Structures*, Mouton & Co., La Haya, Holanda, 1957.

3 [CLIB79] Cline, Ben, "An Introduction to Microprogramming", en *Byte*, abril, 1979.

5 [COHD86] Cohen, Daniel, *Introduction to Computer Theory*, Wiley, Nueva York, 1986.

3 [CRAW86] Cramer, William y Gerry Kane, *68000 Microprocessor Handbook*, 2a. ed., Osborne/McGraw-Hill, Berkeley, 1986.

6 [DAHO72] Dahl, Ole, Edsger Dijkstra y Charles Hoare, *Structured Programming*, Academic Press, Nueva York, 1972.

4 [DATC86] Date, C. J., *Introducción a los sistemas de bases de datos*, Addison-Wesley Iberoamericana, México, 1986.

1 [DAVC78] Davis, N. C. y S. E. Goodman, "The Soviet Bloc's unified system of computers", *Computing Surveys*, Association for Computing Machinery, vol. 10, núm. 3, junio, 1978.

8 [DAVG86] Davis, Gordon y Thomas Hoffmann, *FORTRAN 77: Un estilo estructurado y disciplinado*, McGraw-Hill, México, 1986.

4 [DEIH87] Deitel, H. M., *Introducción a los sistemas operativos*, Addison Wesley Iberoamericana, México, 1987.

7 [DEMT79] De Marco, Tom, *Concise Notes on Software Engineering*, Yourdon Press, Nueva York, 1979.

5 [DEND78] Denning, Peter, Jack Dennis y Joseph Qualitz, *Machines, Languages and Computation*, Prentice Hall, New Jersey, 1978.

4 [DENP71] Denning, Peter, "Third Generation Computer Systems", en *Computing Surveys*, Association for Computing Machinery, vol. 3, núm. 4, diciembre, 1971.

4 [DENP84] Denning, Peter y Robert L. Brown, "Operating Systems", en *Scientific American*, septiembre, 1984.

4 [DEWA84] Dewdney, A. K., "Computer Recreations", en *Scientific American*, julio, 1984.

4 [DONJ72] Donovan, John, *Systems Programming*, McGraw-Hill International, Tokio, 1972.

5 [DOUA70] Dou, Alberto, *Fundamentos de la matemática*, Nueva Colección Labor, núm. 117, Barcelona, 1970.

4 [DOUJ78] Douglas, J. R., "Chess 4.7 versus David Levy: the computer beats a Chess Master", en *Byte*, diciembre, 1978.

8 [FAIR87] Fairley, Richard, *Ingeniería de software*, McGraw-Hill, México, 1987.

1 [FREP86] Freiberger, Paul y Michael Swaine, *Microinformática: orígenes, personajes, evolución y desarrollo*, Osborne/McGraw Hill, Madrid, 1986.

7 [FRIF84] Friedman, Frank y Elliot Koffman, *FORTRAN*, Fondo Educativo Interamericano, México, 1984.

7 [FRIF86] Friedman, Frank y Elliot Koffman, *BASIC*, Addison-Wesley Iberoamericana, México, 1986.

5 [GODK81] Godel, Kurt, *Obras completas*, Alianza Editorial, Col. Alianza Universidad, núm. 286, Madrid, 1981.

1 [GOLH72] Goldstine, H. Herman, *The Computer from Pascal to von Neumann*, Princeton University Press, New Jersey, 1972.

5 [GRAF72] Gracia, Francisco (ed.), *Presentación del lenguaje*, Taurus, núm. 89, Madrid, 1972.

4 [GRAR88] Grauer, Robert y Paul Sugrue, *Aplicaciones de microcomputadoras*, McGraw-Hill, México, 1988.

4 [GRID71] Gries, David, Compiler Construction for Digital Computers, Wiley, Nueva York, 1971.

7 [GRID81] Gries, David, *The Science of Programming*, Springer-Verlag, Nueva York, 1981.

7 [GROP86] Grogono, Peter, *Programación en Pascal*, Addison-Wesley Iberoamericana, México, 1986.

3 [HALI79] Halsema, A. I., "Bubble Memories", en *Byte*, junio, 1979.

7 [HARD87] Harel, David, *Algorithmics: The Spirit of Computing*, Addison-Wesley, Massachusetts, 1987.

3 [HAYJ79] Hayes, John, *Computer Architecture and Organization*, McGraw Hill International, Tokio, 1979.

5 [HEMN69] Hempel, C., E. Nagel, J. Newman et. al., *Matemática, verdad y realidad*, Grijalbo, Barcelona, 1974.

7 [HOAC69] Hoare, Charles, "An axiomatic Basis for Computer Programming", en *Communications of the Association for Computing Machinery*, vol. 12, núm. 10, octubre, 1969.

4 [HOPJ69] Hopcroft, John y Jeffrey Ullman, *Formal Languages and their Relation to Automata*, Addison-Wesley, Massachusetts, 1969.

4 [HOPJ79] Hopcroft, John y Jeffrey Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Massachusetts, 1979.

5 [HOPJ84] Hopcroft, John, "Turing Machines", en *Scientific American*, septiembre, 1984.

3 [INTC87] Intel Corporation, *Microprocessor and Peripheral Hand-book, Volume I: Microprocessor*, Intel Corporation, California, 1987.

7 [JACM75] Jackson, M. A., *Principles of Program Design*, Academic Press, Londres, 1975.

8 [JENK74] Jensen, Kathleen y Niklaus Wirth, *Pascal User Manual and Report*, Springer-Verlag, Nueva York, 1974.

8 [JONW82] Jones, William, *Programming Concepts, A Second Course, with Examples in Pascal*, Prentice-Hall, New Jersey, 1982.

4 [KERB76] Kernighan, Brian y P. J. Plauger, *Software Tools*, Addison Wesley, Massachusetts, 1976.

7 [KERB78] Kernighan, Brian y P. Plauger, *The Elements of Programming Style*, McGraw-Hill, Nueva York, 1978.

8 [KERB81] Kernighan, Brian y P. J. Plauger, *Software Tools in Pascal*, Addison-Wesley, Massachusetts, 1981.

7 [KERB85] Kernighan, Brian y Dennis Ritchie, *El lenguaje de programación C*, Prentice-Hall, México, 1985.

5 [KLIM72] Kline, Morris, *Mathematical Thought from Ancient to Modern Times*, Oxford University Press, Nueva York, 1972.

7 [KNUD73] Knuth, Donald, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Massachusetts, 1973.

8 [KOFE86] Koffman, Elliot, *Pascal. Introducción al lenguaje y resolución de problemas con programación estructurada*, Addison Wesley Iberoamericana, México, 1986.

3 [KUHL79] Kuhn, Larry y Robert Myers, "Ink-jet printing", en *Scientific American*, abril, 1979.

7 [LEVG80] Levine, Guillermo, *Manual para el usuario del sistema FOREST (FORtran ESTructurado)*, Departamento de ingeniería, Universidad Autónoma Metropolitana-Iztapalapa, México, 1980.

1 [LEVR82] Levine, Ronald, "Supercomputers", en *Scientific American*, enero, 1982.

6 [LINR79] Linger, Richard, Harlan Mills y Bernard Witt, *Structured Programming, Theory and Practice*, Addison-Wesley, Massachusetts, 1979.

4 [LISM85] Lister, M. A., *Fundamentals of Operating Systems*, Macmillan, Londres, 1985.

7 [LOGI70] *Lógica*, National Council of Teachers of Mathematics, Col. Temas de matemáticas, vol. 12, Trillas, México, 1970.

3 [LUCH76] Lucas, Henry, Jr., *The Analysis, Design and Implementation of Information Systems*, McGraw-Hill International, Tokio, 1976.

1 [LUKH79] Lukoff, Herman, *From Dits to Bits. A Personal History of the Electronic Computer*, Robotics Press, Oregon, 1979.

4 [MADS74] Madnick, Stuart y John Donovan, *Operating Systems*, McGraw Hill, Nueva York, 1974.

6 [MCGC75] McGowan, Clement y John Kelly, *Top-Down Structured Programming Techniques*, Petrocelli/Charter, Nueva York, 1975.

5 [MINM67] Minsky, Marvin, *Computation: Finite and Infinite Machines*, Prentice-Hall, New Jersey, 1967.

4 [MORC84] Morgan, Christopher y Mitchell Waite, *Introducción al microprocesador 8086/8088*, Byte/McGraw-Hill, México, 1984.

3 [NEWW81] Newman, William y Robert Sproull, *Principles of Interactive Computer Graphics*, McGraw-Hill International, Tokio, 1981.

7 [ORRK77] Orr, Kenneth, *Structured Systems Development*, Yourdon Press, Nueva York, 1977.

3 [PATD83] David Patterson, "Microprogramming", en *Scientific American*, marzo, 1983.

4 [PAVR81] Pavelle, Richard, M. Rothstein y J. Fitch, "Computer Algebra", en *Scientific American*, diciembre, 1981.

4 [PETJ83] Peterson, James y Abraham Silberschatz, *Operating System Concepts*, Addison-Wesley, Massachusetts, 1983.

7 [PHIA87] Philippakis, Andreas y Leonard Kazmier, *COBOL estructurado*, McGraw-Hill, Colombia, 1987.

5 [PLAT73] Platón, *Diálogos*, Porrúa, México, 1973.

1 [PYLZ75] Pylyshyn, Zenon (ed.), *Perspectivas de la revolución de los computadores*, Alianza Editorial, Col. Alianza Universidad, núm. 119, Madrid, 1975.

1 [RALA76] Ralston, Anthony y C. L. Meek, (eds.), *Encyclopedia of Computer Science*, Petrocelli/Charter, Nueva York, 1976.

1 [RUSB86] Russell, Bertrand, *Los problemas de la filosofía*, Nueva Colección Labor, Barcelona, 1986.

6 [SAMJ69] Sammet, Jean, *Programming Languages: History and Fundamentals*, Prentice-Hall, New Jersey, 1969.

8 [SCHW74] Schick, William y Charles Merz, *FORTRAN IV para ingeniería*, McGraw-Hill, México, 1974.

4 [SCIA71] Scientific American, *Computers and Computation*, W. H. Freeman, San Francisco, 1971.

3 [SIED82] Siewiorek, Daniel, Gordon Bell y Allen Newell, *Computer Structures: Principles and Examples*, McGraw-Hill International, Tokio, 1982.

1 [SLAR87] Slater, Robert, *Portraits in Silicon*, MIT Press, Massachusetts, 1987.

8 [SOMI88] Sommerville, Ian, *Ingeniería de Software*, Addison-Wesley Iberoamericana, México, 1988.

7 [SUPP66] Suppes, Patrick, *Introducción a la lógica simbólica*, CECSA, México, 1966.

4 [TANA87] Tanenbaum, Andrew, *Operating Systems: Design and Implementation*, Prentice-Hall, New Jersey, 1987.

7 [TARA68] Tarski, Alfred, *Introducción a la lógica y a la metodología de las ciencias deductivas*, Nueva Ciencia - Nueva Técnica, Espasa Calpe, Madrid, 1968.

6 [TESL84] Tesler, Lawrence, "Programming Languages", en *Scientific American*, septiembre, 1984.

6 [TREJ82] Tremblay, Jean-Paul y Richard Bunt, *Introducción a la ciencia de las computadoras, enfoque algorítmico*, McGraw-Hill, México, 1982.

4 [TREJ85] Tremblay, Jean-Paul y Paul G. Sorensen, The Theory and Practice of Compiler Writing, McGraw-Hill, Nueva York, 1985.

6 [TUCA87] Tucker, Allen, *Lenguajes de programación*, McGraw-Hill, México, 1987.

4 [ULLJ76] Ullman, Jeffrey, *Fundamental Concepts of Programming Systems*, Addison-Wesley, Massachusetts, 1976.

4 [ULLJ82] Ullman, Jeffrey, *Principles of Database Systems*, Computer Science Press, Maryland, 1982.

3 [VACA75] Vacroux, André, "Microcomputers", en *Scientific American*, mayo, 1975.

7 [WEIG71] Weinberg, Gerald, *The Psychology of Computer Programming*, Van Nostrand Reinhold, Nueva York, 1971.

3 [WHIR80] White, Robert, "Disk Storage Technology", en *Scientific American*, agosto, 1980.

4 [WIEG83] Wiederhold, Gio, *Diseño de bases de datos*, McGraw-Hill, México, 1983.

4 [WINP84] Winston, Patrick, *Artificial Intelligence*, Addison-Wesley, Massachusetts, 1984.

4 [WINT84] Winograd, Terry, "Computer Software for Working with Language", en *Scientific American*, septiembre, 1984.

7 [WIRN71] Wirth, Niklaus, "Program Development by Stepwise Refinements", en *Communications of the Association for Computing Machinery*, vol. 14, núm. 4, abril, 1971.

7 [WIRN76] Wirth, Niklaus, *Algorithms + Data Structures = Programs*, Prentice-Hall, New Jersey, 1976.

6 [WIRN84] Wirth, Niklaus, "Data Structures and Algorithms", en *Scientific American*, septiembre, 1984.

8 [YOUE79] Yourdon, Edward y Larry Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, New Jersey, 1979.

La siguiente lista corresponde a los libros y artículos (casi siempre de nivel avanzado) que se mencionaron en el texto, no como referencias directas para el material del capítulo, sino como fuentes adicionales de consulta o estudio. El número entre paréntesis indica el capítulo en el que fueron mencionados.

Artículos

(6) Bohm, Conrado y Giuseppe Jacopini, "Flow Diagrams, Turing Machines and Languages With Only Two Formation Rules", *Communications of the Association for Computing Machinery*, vol. 9, núm. 5, mayo, 1966.

(5) Brainerd, J. Charles, "The Origins of Number Concepts", *Scientific American*, marzo, 1973.

(6) Dijkstra, Esdger, carta al editor, *Communications of the ACM*, marzo, 1968.

(2) Edwards, M. Harold, "Fermat's Last Theorem", *Scientific American*, octubre, 1978.

(4) Earley, Jay, "An Efficient Context-Free Parsing Algorithm", *Communications of the ACM*, febrero, 1970.

(6) Harel, David, "On Folk Theorems", *Communications of the ACM*, vol. 23, núm. 7, julio, 1980.

(7) Higgins, David, "Structured Programming with Warnier-Orr Diagrams", *Byte*, diciembre de 1977 (Parte I), enero de 1978 (Parte II).

(4) Knuth, Donald, "On the translation of languages from left to right", *Information and Control*, octubre, 1965.

(6) Knuth, Donald, "Structured programming with go to statements", *Computing Surveys of the ACM*, vol. 6, núm. 4, 1974.

(4) Revista *Byte*, número de abril de 1985, dedicado a la inteligencia artificial.

(4) Revista *Scientific American*, número de octubre de 1987, dedicado a las nuevas generaciones de computadoras.

(A) Ritchie, Dennis y Ken Thompson, "The UNIX Time-Sharing System", *Communications of the ACM*, julio, 1974.

(1) Williams, M. James, "Antique Mechanical Computers", (publicado en tres partes), *Byte*, julio-septiembre, 1978.

(4) Zobrist, Albert y Frederic Carlson, "An Advice-Taking Chess Computer", *Scientific American*, junio, 1973.

Libros

(A) Anderson, Gail y Paul.Anderson, *The Unix C Shell Field Guide*, Prentice Hall, New Jersey, 1986.

(A) *ACM Turing Award Lectures: The First Twenty Years*, 1966-1985, Addison Wesley, Massachusetts, 1987.

(A) Blackburn, Lawrence y Marcus Taylor, *UNIX. Guía de bolsillo*, Fondo Educativo Interamericano, México 1986.

(5) Boole, George, *An Investigation on the Laws of thought on which are founded the Mathematical Theories of Logic and Probabilities*, 1854, en edición facsimilar, Dover, Nueva York, 1958.

(A) Christian, Kaare, *The Unix Operating System*, Wiley, Nueva York, 1983.

(B) Feuer, Alan, *The C Puzzle Book*, Prentice-Hall, New Jersey, 1982.

(A) Fiedler, David y Bruce Hunter, *Unix System Administration*, Hayden, New Jersey, 1986.

(B) Gehani, Narain, *Advanced C: Food for the Educated Palate*, Computer Science Press, Maryland, 1985.

(B) Gehani, Narain, *C: An Advanced Introduction*, Computer Science Press, Maryland, 1985.

(B) Hancock, Les y Morris Krieger, *The C Primer*, McGraw-Hill, Nueva York, 1986.

(2) Hamacher, Vranesic y Zaky, *Organización de computadoras*, McGraw-Hill, México, 1987.

(7) Jackson, M. A., *Systems Development*, Prentice-Hall, New Jersey, 1983.

(B) Kelley, Al e Ira Pohl, *Lenguaje C*, Addison-Wesley Iberoamericana, México, 1987.

(A) Kernighan, Brian y Rob Pike, *El entorno de programación Unix*, Prentice Hall Hispanoamericana, México, 1987.

(3) Knuth, Donald, *Computers and Typesetting*, serie de libros publicada por Addison-Wesley, Massachusetts, 1984.

(5) Lukasiewicz, Jan, *Aristotle's Syllogistic from the Standpoint of Modern Formal Logic*, Oxford University Press, Nueva York, 1951.

(A) McGilton, Henry y Rachel Morgan, *Introducing the Unix System*, McGraw Hill, Nueva York, 1983.

(2) Perelmán, Yakov, *Álgebra recreativa*, Editorial Mir, Moscú, 1978.

(6) Peter, Laurence y Raymond Hull, *El principio de Peter*, Plaza & Janés, Barcelona, 1971.

(B) Plum, Thomas, *Learning to Program in C*, Plum Hall, New Jersey, 1983.

(B) Plum, Thomas y Jim Brodie, *Efficient C*, Plum Hall, New Jersey, 1985.

(B) Purdum, J., *C Programming Guide*, QUE Corporation, Indiana, 1983.

(B) Schwaderer, David, *C Wizard's Programming Reference*, Wiley & Sons, Nueva York, 1985.

(A) Silvester, Peter, *The Unix System Guidebook: An Introductory Guide for Serious Users*, Springer-Verlag, Nueva York, 1984.

(A) Sobell, Mark, *Guía práctica para el sistema operativo Unix*, Addison Wesley Iberoamericana, México, 1987.

(A) Thomas, Rebecca y Jean Yates, *A User Guide to the Unix System*, Osborne/McGraw-Hill, Berkeley, 1982.

(B) Tondo, Clovis y Scott Gimpel, *The C Answer Book*, Prentice-Hall, New Jersey, 1985.

(B) Wagner-Dobler, Friedman, *Lenguaje C. Guía práctica*, Addison-Wesley Iberoamericana, México, 1987.

(B) Waite, Mitchell, Stephen Prata y Donald Martin, *C Primer Plus*, Howard W. Sams & Co., Indiana, 1984.

(B) Ward, Terry, *Applied Programming Techniques in C*, Scott, Foresman & Co., Illinois, 1985.

(5) Weizenbaum, Joseph, *Computer Power and Human Reason*, W. H. Freeman, San Francisco, 1976.

(5) Wittgenstein, Ludwig, *Tractatus Logico-Philosophicus*, Alianza Editorial, Col. Alianza Universidad, núm. 50, Madrid, 1973.

(A) Wood, Patrick y Stephen Kochan, *Unix System Security*, Hayden, New Jersey, 1985.

Índice temático

D

E