Cristóbal Pareja
Ángel Andeyro
Manuel Ojeda

Introducción a la Informática

I. Aspectos generales

1ª Edición
Febrero 1994

ISBN: 84-7491-489-2
Depósito Legal: M-7713-94

Índice General

Presentación

Hay muchos y excelentes libros sobre informática en un nivel introductorio, con diversos enfoques y escritos en nuestra lengua. Muchos de ellos están dirigidos a futuros informáticos, por lo que resultan, quizá, demasiado profundos para quienes sólo persiguen aplicar la informática en su trabajo; muchos otros, en cambio, se dirigen a no profesionales, por lo que ofrecen una visión panorámica demasiado general de diversos aspectos de la informática, sin detenerse a analizar las implicaciones prácticas de esos aspectos.

Este libro se sitúa entre ambos extremos, ofreciendo un medio para introducir en la informática a profesionales de otros campos que, sin embargo, necesiten aplicar adecuadamente los computadores en su trabajo, poniendo en marcha programas de aplicación o manejando bancos de datos e incluso, muchas veces, resolviendo por sí mismos problemas no contemplados en los programas comercializados.

Así pues, el contenido del texto incluye los temas que consideramos básicos para una introducción práctica a la informática. Quizá sea este enfoque práctico lo que mejor distinga a este texto de otros de introducción a la informática, que adolecen muchas veces de contenidos muy extensos y teóricos, casi enciclopédicos, en los que es difícil diferenciar los conocimientos útiles en general de aquéllos que sólo encontrarán de utilidad informáticos profesionales.

Por lo tanto, este libro se dirige a quienes necesiten una formación introductoria en informática, con un enfoque básico y práctico, pero riguroso. Entre ellos se encuentran, en primer lugar, todos aquellos alumnos

de los primeros cursos universitarios que cuenten con asignaturas sobre informática, tanto si se utiliza como herramienta aplicada a la resolución de problemas como si es el propio objeto de estudio.

Este libro se dirige asimismo a aquellas personas que, de una u otra forma, están relacionadas con el mundo de la informática (operadores, comerciales, empresarios, etc.) y, sin embargo, ignoran lo que se esconde tras la fachada de un computador (cómo se almacena la información, cuáles son los procesos que se están ejecutando en su interior, etc.) y que utilizan una jerga técnica sin entender bien de qué hablan.

Por último, este libro se dirige también a todos los usuarios de computadores que estén interesados en conocer mejor su herramienta de trabajo, en saber qué estan haciendo y por qué lo hacen. El desarrollo y abaratamiento de los sistemas informáticos hace que computadores de gran potencia que antes sólo se encontraban en grandes centros de cálculo, atendidos por administradores de sistemas cualificados y dedicados por completo a dicha tarea, estén hoy sobre nuestra mesa, y que el usuario se tenga que encargar de administrar los recursos de su computador, por ejemplo, gestionando la memoria, haciendo copias de seguridad, eliminando virus, etc. Por todo ello, cualquier usuario debería dedicar siquiera una pequeña parte de su tiempo a conocer los principios básicos de funcionamiento y gestión de su computador.

A todos ellos, este texto les ofrece la posibilidad de adquirir esos conocimientos, partiendo de cero, o de actualizarlos.

Se ha seleccionado el contenido partiendo de las directrices señaladas en [DCG*89] y [ACM91] aunque, en un nivel introductorio, no sea posible ni deseable abarcar todos los temas que sólo interesan al futuro profesional de la informática. Así por ejemplo, quedan completamente fuera del alcance de nuestro texto las áreas de *inteligencia artificial y robótica* (*AI*) o la *comunicación hombre-máquina* (*HU*).

Es frecuente, en cambio, que los destinatarios mencionados necesiten una parte de conocimientos generales sobre informática y otra sobre desarrollo de algoritmos en un lenguaje de alto nivel: ambas partes se incluyen en los programas de asignaturas de introducción a la informática

de los primeros cursos universitarios, y también interesan esas dos partes a los profesionales que usen la informática como una herramienta y deseen sacar partido de ella.

El presente texto comprende dos volúmenes, complementarios, dedicados respectivamente a esas dos partes, separando así la presentación de los *conceptos generales* y el desarrollo y organización de *algoritmos y estructuras de datos*. En la primera parte, el enfoque práctico nos ha llevado a relacionar los contenidos estudiados con sus repercusiones prácticas o su utilización. En la segunda, se ha unificado el estudio de los algoritmos con su desarrollo, haciendo uso de un lenguaje de programación concreto.

Este primer volumen se ha dividido en siete capítulos, de los cuales el primero ofrece una visión panorámica de la informática, de los computadores y de su uso en la actualidad: bien poniendo en marcha aplicaciones ya desarrolladas y adaptándolas a nuestras necesidades, o preparando soluciones para problemas nuevos. Se presentan asimismo las primeras aproximaciones a los conceptos de algoritmo, programación y lenguajes de programación.

En el capítulo 2 se aborda la representación digital de la información. Conociendo las distintas formas de representación, el programador podrá elegir las más apropiadas a las características y naturaleza de su problema y prever sus limitaciones. Por otra parte, el usuario de aplicaciones tendrá una idea aproximada de cómo se almacena su información, el espacio que ocupa y las circunstancias en que los resultados podrían no ser del todo fiables.

El capítulo 3 está dedicado al aspecto físico de los computadores y sus periféricos. Se explica su funcionamiento introduciendo los lenguajes de bajo nivel, y se comentan brevemente las arquitecturas orientadas al procesamiento en paralelo.

En el capítulo 4 se estudian los conceptos básicos para entender lo que son los sistemas operativos, cuáles son sus funciones y cómo las llevan a cabo.

El capítulo 5 se dedica a los lenguajes de programación, en espe-

cial los lenguajes evolucionados. Se tratan distintos modelos de programación, además del imperativo, de amplia difusión en estos años. Se introducen los metalenguajes para describir su sintaxis por su utilidad, tanto a programadores como a simples usuarios de sistemas operativos, incluso en el nivel de los comandos. Finalmente, se estudian los distintos tipos de traductores y los entornos de programación.

El capítulo 6 trata sobre las bases de datos, una de las aplicaciones de mayor aplicación en el mundo empresarial. Entre los modelos desarrollados, destacamos el relacional y, entre los lenguajes de consulta, el SQL, debido a la extensa difusión de ambos en la actualidad.

Hemos dedicado el capítulo 7 a los orígenes, estado actual y perspectivas de futuro de la informática. Aunque muchos textos sitúan este tema al principio, hemos preferido estudiarlo una vez que se conocen los conceptos y la terminología básicos. De esta forma mejora la comprensión del capítulo y puede valorarse en su justa medida cada uno de los logros históricos que en él se recogen.

Debido al enfoque práctico que perseguimos con este libro, se incluyen dos apéndices dedicados a introducir en el manejo de dos sistemas operativos concretos de gran difusión en la actualidad: el DOS y el UNIX. El hecho de traer aquí estos anexos responde a dos razones: por un lado, conocer las *características* de cada sistema interesa al estudiante como un ejemplo particular del capítulo 4, permitiendo ver cómo se lleva a la práctica lo estudiado en la teoría; por otro, siempre resulta de utilidad al principiante disponer de un pequeño prontuario de las *órdenes* o mecanismos más inmediatos que necesitará sin duda para empezar a desenvolverse en el entorno de esos sistemas operativos.

En la mayoría de los capítulos, se ha incluido una pequeña colección de cuestiones y sencillos ejercicios de aplicación, útiles para afianzar los conceptos introducidos.

Además, en cada capítulo se han seleccionado unas pocas referencias, para completar los contenidos presentados con otros enfoques, o bien para profundizar en el tema.

Agradecimientos

En primer lugar, es inexcusable agradecer a la Editorial la confianza que ha puesto en nosotros al aceptar una publicación sobre un tema en el que, ya lo hemos dicho, existen abundantes textos en nuestra lengua, así como su paciencia en la recepción de los originales.

También debemos dejar constancia de nuestra gratitud hacia los compañeros que nos han alentado, desde el principio, a redactar este trabajo. En especial, a Benjamín Hernández Blázquez, María Ángeles Medina Sánchez, Salvador Paz Martínez, Inma Pérez de Guzmán Molina y Marisol Timoneda Salinas, y también a todos aquellos alumnos que, repetidamente, nos han sugerido la compilación de unos apuntes de clase.

Durante la redacción de este trabajo, se han recogido numerosas opiniones y sugerencias. En particular, debemos agraceder las minuciosas revisiones y comentarios hechos por Manuel Enciso García-Oliveros, Carlos Rossi Jiménez, José Luis Galán García, Jaime Fernández Martínez, Mª Ángeles Cano Colorado, Óscar Martín Sánchez y Cristina Rodríguez Iglesias.

Finalmente, quisiéramos pedir la colaboración de los lectores para subsanar las posibles deficiencias que encuentren.

Capítulo 1

Conceptos Básicos

El objetivo principal de este capítulo consiste en ofrecer una visión general del contenido del libro, presentando los conceptos generales y los términos más usados en informática. Esta primera aproximación nos permite situar cada uno de los temas siguientes en relación con los demás, y dentro del contexto de la informática.

1.1 Informática

La *informática* es la ciencia que estudia el procesamiento automático de la información. Aunque la necesidad de razonar sobre este tipo de procesos existe desde tiempo atrás, la consolidación de la informática como ciencia sólo se produce con el desarrollo de los computadores, a partir de los años cuarenta. Se trata, por lo tanto, de una ciencia muy joven, pero que ha evolucionado a gran velocidad.

La piedra maestra sobre la cual se ha podido desarrollar la informática la representa el *computador*, que es una herramienta de gran eficacia en muy diversos trabajos, y en particular en aquéllos que manejan un gran volumen de datos o de operaciones. Esta versatilidad tiene dos aspectos: por un lado, es posible usarlo como herramienta para aplicaciones concretas ya desarrolladas (1.4), y por otro se pueden diseñar soluciones a la medida de problemas nuevos, mediante la programación (1.5).

El desarrollo de un programa nuevo para resolver un determinado problema requiere, por una parte, conocer algún procedimiento sistemático (*algoritmo*) que lleve a su solución, y por otra, la necesidad de expresarlo en un *lenguaje de programación* que el computador pueda comprender y ejecutar.

1.2 Computador

Un *computador* es una máquina electrónica que procesa información siguiendo las instrucciones de un programa registrado.

Para comunicarse con el exterior dispone de unos medios de entrada, a través de los que recibe la información, y unos medios de salida, por donde la envía. Tiene dispositivos que le permiten almacenar la información (los datos, los resultados y el propio programa) y procesarla siguiendo las instrucciones del programa.

La información que se procesa en el computador (programas, datos y resultados) está expresada en forma *digital* binaria, combinando ceros y unos. En consecuencia, tanto los programas como los datos y resultados deben codificarse en este formato para poder ser procesados. Una vez obtenidos los resultados, éstos tienen que ser decodificados para mostrarlos al usuario.

Como hemos visto, un computador se compone de dos partes claramente diferenciadas: una física, que podemos tocar, constituida por circuitos electrónicos, teclado, pantalla, unidades de disco, etc., llamado *hardware*, o en castellano *soporte físico*, y otra parte inmaterial, que no

usuario
software de aplicaciones y del sistema
sistema operativo
hardware

Tabla 1.1. Estructura de niveles en un computador.

podemos tocar, constituida por los programas y datos, llamada *software* en inglés y *soporte lógico* en castellano. Ambas partes están íntimamente relacionadas de forma que una no puede operar sin la otra y viceversa.

1.3 Sistema operativo

Cuando se pone en marcha el computador, el primer programa que entra en funcionamiento es el *sistema operativo*, que gestiona y coordina los dos aspectos, físico y lógico, del computador. Se trata de un conjunto de programas que se interrelacionan estrechamente con el hardware, gestionando los procesos en ejecución, las operaciones de entrada y salida y la memoria. Por ello, resulta imprescindible para el funcionamiento del computador.

Los demás programas funcionan sobre el sistema operativo, y son gestionados por él. Entre ellos, se encuentran las herramientas para el desarrollo de programas (tales como los editores y traductores de lenguajes), y también los programas de aplicaciones.

Por lo tanto, podemos decir que dentro del computador existe cierta organización por niveles (véase la tabla 1.1): en el nivel más bajo se encuentra el hardware, que por sí mismo no puede realizar ninguna tarea; a continuación se encuentra el sistema operativo, y desde él se arrancan los otros programas, que a su vez se relacionan directamente con el usuario.

En resumen, el hardware no puede funcionar por sí mismo: necesita la ayuda del software. La unión de ambos constituye una máquina virtual, tremendamente versátil.

1.4 Aplicaciones

El software de aplicaciones está formado por aquellos programas que han sido desarrollados para realizar tareas concretas. Se llama así porque el computador "se aplica" a un trabajo determinado, facilitando su ejecución y resolución. Por ejemplo, un procesador de textos, una hoja de cálculo, un gestor de bases de datos, un generador de gráficos, un programa de contabilidad, juegos, etc.

Entre las aplicaciones más utilizadas se encuentran las siguientes:

- Los *procesadores de textos* son programas que facilitan la elaboración de textos en el computador, desde una carta hasta un libro. Permiten operar con márgenes, tabuladores, justificación, sangrado, tipos de letra, búsqueda y sustitución de palabras, paginación, separación de sílabas, sinónimos, ortografía, etc. Son probablemente los programas más usados.

- Las *hojas de cálculo* son programas utilizados en la creación de tablas, con datos relacionados entre sí, inicialmente ideados para el análisis financiero. Tienen un formato matricial, en el que se pueden definir operaciones y funciones sobre las distintas componentes de la matriz. Al modificar algún dato, todas las operaciones que lo utilizan son actualizadas de forma automática.

- Los *gestores de bases de datos* permiten gestionar la información referida a personas o artículos, realizando operaciones de edición, ordenación, búsqueda, etc.

- Los *generadores de gráficos* facilitan la creación de distintos tipos de gráficos, a partir de datos de hojas de cálculo o bases de datos, o directamente introducidos por el usuario.

Existen muchas otras aplicaciones, de uso menos general, como son los programas matemáticos, estadísticos, de CAD (Diseño Asistido por Computador), aplicaciones contables y de gestión de empresas, comunicaciones, juegos, etc.

El desarrollo del software de aplicaciones ha sido muy grande (se calcula que para los computadores compatibles con IBM hay más de cien mil aplicaciones diferentes). A su vez, los distintos programas van evolucionando, y aparecen nuevas versiones con más posibilidades y mayor velocidad de ejecución, aunque también con mayor demanda de potencia y memoria.

En algunos casos, varios de estos programas se unen en uno solo, compartiendo datos e instrucciones, y constituyen un *paquete integrado*. En general suele integrarse una hoja de cálculo con un generador de gráficos y, a veces, con una base de datos y un procesador de textos.

Gran parte de los programas de aplicación pueden configurarse, en mayor o menor medida, de acuerdo con los gustos y necesidades del usuario; sin embargo, es posible que un programa concreto no pueda atender esas necesidades. En este caso, resulta muy difícil, por no decir imposible (e ilegal en muchos casos), modificar el programa para incluir una nueva tarea. Conscientes de esta falta de flexibilidad del software, muchos fabricantes están presentando programas de aplicación que a su vez pueden ser programados, mediante lenguajes de programación propios o estándares.

Así pues, aunque la mayoría de las personas que utilizan los computadores trabajan con programas de aplicación y no necesitan recurrir a la programación, también hay un buen número de usuarios que, sin ser informáticos profesionales, pueden obtener un mayor rendimiento de estos programas a través de la programación. Por ejemplo, muchas hojas de cálculo, gestores de bases de datos y paquetes matemáticos, hoy en día, son programables.

1.5 Algoritmos y programas

El desarrollo de programas es otro de los aspectos fundamentales de la utilización de los computadores porque continuamente aparecen nuevos problemas o tareas susceptibles de ser procesados de forma automática. Al mismo tiempo se van detectando las lagunas o deficiencias en las aplicaciones existentes, lo que impulsa a su renovación con la

creación de nuevas versiones de los programas existentes. El desarrollo del hardware posibilita también la aparición de nuevas aplicaciones más potentes y con mayores demandas de recursos.

Desde el planteamiento de un problema hasta la obtención de su solución en el computador hay que recorrer una serie de etapas:

1. En primer lugar, antes de resolver un problema en el computador hay que conocer los pasos y operaciones que hay que realizar para obtener la solución del problema, es decir, su algoritmo, porque el computador solamente es capaz de seguir aquellas instrucciones que nosotros le indiquemos. Si no conocemos el proceso que nos conduce a la solución del problema, el computador no nos la va a dar. Esta secuencia de pasos y operaciones constituye una solución general al problema planteado de forma que, siguiendo el proceso, se llega a la solución del problema sean cuales fueran los datos proporcionados.

2. Una vez conocida esta solución general del problema, hay que expresarla en un lenguaje especial, que pueda ser comprendido y ejecutado por el computador, es decir, en un lenguaje de programación, creando un programa.

3. Posteriormente, hay que comprobar que el programa produce las soluciones esperadas (ya sea utilizando datos de prueba o mediante métodos formales) y subsanar los errores detectados.

4. Por último, es importante documentar el programa de forma que si cambiaran algunas de las circunstancias iniciales, sea posible modificar y adaptar convenientemente, facilitando las labores de mantenimiento.

1.5.1 Algoritmos

Una de las características de los seres humanos es su capacidad para plantearse y resolver problemas. Éstos pueden ser de naturaleza muy

diversa, desde los problemas más inmediatos relacionados con la propia subsistencia, hasta los problemas más abstractos de naturaleza matemática o filosófica.

Un *algoritmo* es la descripción precisa de los pasos que nos permiten obtener la solución de un problema determinado. En general, los pasos son acciones u operaciones que se efectúan sobre ciertos objetos. Al comienzo del algoritmo, los objetos tienen unos valores iniciales (los datos) que varían como consecuencia del proceso descrito por el algoritmo, obteniéndose los valores de salida o resultados.

La informática estudia el procesamiento de la información mediante algoritmos, aunque el concepto de algoritmo, que proviene de las matemáticas, es muy anterior e independiente de la existencia de la informática y los computadores. [1]

El concepto de algoritmo tiene una importancia fundamental dentro de la informática, por ser previo a la resolución del problema en el computador; si no se conoce el algoritmo para resolver un problema, no puede plantearse su resolución en el computador.

Aunque existen algoritmos registrados para la realización de tareas muy variadas, en general los algoritmos desarrollados resuelven sólo determinadas partes de un problema como, por ejemplo, la ordenación de una lista de valores, pero no un problema real completo. En consecuencia, habrá que diseñar un algoritmo para su resolución.

El diseño de algoritmos implica un análisis profundo del problema, de sus datos iniciales, del proceso que se les aplica y de los resultados esperados. A partir de este análisis debe establecerse cuál es la mejor estructura de datos para resolver el problema. De hecho, Niklaus Wirth, uno de los padres de la programación estructurada, titula una de sus obras fundamentales *Algoritmos + Estructuras de Datos = Programas*, mostrando la importancia que concede a dichas estructuras [Wir86b].

[1] Se conoce un algoritmo para el cálculo del máximo común divisor de dos números naturales, debido a Euclides, que data del siglo IV a.C., al que se conoce como el "abuelo" de todos los algoritmos.

Existen técnicas que facilitan el diseño de algoritmos, tales como la programación estructurada, la programación modular, el refinamiento por pasos, el diseño descendente y la estructuración y abstracción de los datos.

Para poder expresar algoritmos se utilizan lenguajes con la necesaria precisión, llamados lenguajes algorítmicos, que son independientes de los lenguajes de programación. Los más utilizados son el seudocódigo y los diagramas de flujo.

Dado un problema concreto y conocido el algoritmo que lo resuelve, para obtener la solución del problema tenemos que partir de los datos de entrada, y ejecutar las acciones descritas en el algoritmo. Al proceso de ejecutar un algoritmo concreto para unos datos determinados se le llama *cómputo*, de donde procede el término computador. El *procesador* es quien ejecuta materialmente el cómputo.

1.5.2 Programación

Como decíamos al principio, consideramos al computador como una herramienta que nos ayuda en la resolución de problemas; para ello es preciso conocer previamente un algoritmo que lleve a su solución. A continuación hay que expresar el algoritmo en un lenguaje de programación, que pueda ser comprendido y ejecutado por el computador, desarrollándose un programa. A este proceso se le llama *programación*.

Una aportación sustancial para realizar este paso tan delicado con la corrección necesaria la constituyen:

1. El refinamiento por pasos, que permite aumentar el grado de detalle en la expresión del algoritmo según convenga, para adaptarlo a las necesidades del lenguaje.

2. La programación estructurada, que utiliza las estructuras de programación propias de los lenguajes evolucionados.

3. La programación modular que, al permitir el uso de módulos o subprogramas, facilita el empleo de otras técnicas de diseño de algoritmos y la depuración de los programas .

4. No debe olvidarse la importancia de la estructuración y abstracción de datos, presente también en los lenguajes evolucionados, que debe emplearse con todo su potencial en el diseño de algoritmos y, posteriormente, en los programas.

El desarrollo de aplicaciones cada vez más complejas y el crecimiento del sector de producción de software, ha hecho que se apliquen a la programación técnicas de ingeniería que garanticen la viabilidad y calidad de los grandes proyectos de aplicaciones, lo que se conoce como *ingeniería del software*.

1.5.3 Lenguajes de Programación

El computador dispone de un conjunto de instrucciones que son reconocidas y ejecutadas por el procesador. Estas instrucciones se expresan, al igual que los datos, en forma digital binaria, si bien para reconocerlas mejor y evitar errores se les asignan unos nombres mnemotécnicos que permiten recordar sus funciones. Estas instrucciones constituyen el *lenguaje de máquina* del computador, y suelen ser diferentes en función del fabricante del procesador.

El lenguaje de máquina es ejecutado a gran velocidad por el procesador, en los computadores actuales esta velocidad se mide en millones de operaciones por segundo; por otra parte, el lenguaje de máquina permite el acceso directo a todos los órganos del computador. Por estos motivos el lenguaje máquina es insustituible en aquellas aplicaciones donde sea necesaria una gran rapidez de ejecución, o el acceso directo a ciertos órganos del computador.

Las instrucciones del lenguaje de máquina son en general muy poco potentes, operan sobre datos de pequeño tamaño, y en muchos casos no incluyen ni multiplicaciones ni divisiones. Para poder operar sobre datos mayores o realizar operaciones más complejas, tales como potencias o logaritmos, hay que fraccionar los datos y aplicar sucesivamente las operaciones simples, siguiendo algoritmos específicos. Por este motivo, y por su estrecha relación con el hardware, a los lenguajes de máquina se les llama *lenguajes de bajo nivel*. En consecuencia, la tarea de escribir

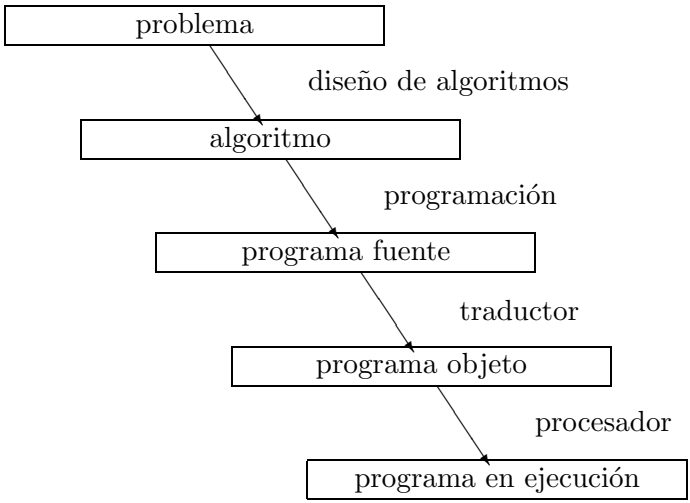


Figura 1.1. Resolución de un problema mediante la programación.

programas en lenguaje de máquina es tediosa y está sujeta a muchos errores.

Ante la necesidad de escribir programas cada vez más complejos y fiables, los informáticos desarrollaron lenguajes con niveles más elevados de abstracción, que incluían operaciones y datos más complejos a los que llamaron *lenguajes de alto nivel*. De esta forma se acorta el camino entre el algoritmo y su expresión en forma de programa, porque los lenguajes de alto nivel pueden expresar mejor las acciones y objetos que intervienen en los algoritmos.

Una de las grandes ventajas de estos lenguajes radica en que la traducción del programa escrito en lenguaje de alto nivel (*programa fuente*) al lenguaje de máquina (*programa objeto*) que, como recordamos, es el único que puede ser ejecutado por el computador, es automática y se realiza por un programa traductor.

Para ello, junto con las especificaciones del lenguaje, reglas de sintaxis y semántica, se desarrollan los necesarios programas de traducción. Durante el proceso de traducción se suelen detectar errores, debidos al

incumplimiento de las reglas sintácticas del lenguaje o a causas más sutiles, que deben corregirse antes de poder ejecutar el programa. Si durante la ejecución de un programa se realizan operaciones no permitidas (tales como divisiones por cero, accesos fuera de límites, ...) se producen los llamados errores *de ejecución*.

Aunque un programa se haya traducido eliminando todos los errores sintácticos y de ejecución, aún puede no realizar correctamente la tarea para la que fue creado, por contener errores *lógicos*. Por lo tanto es necesario comprobar el funcionamiento del programa utilizando datos de prueba que permitan realizar los cálculos a mano y comprobar así las partes más conflictivas del programa. Este proceso se conoce como *depuración* de los programas.

En la actualidad la mayoría de los programas se escriben en lenguajes de alto nivel, reservándose los lenguajes de bajo nivel para aquellas aplicaciones donde se necesite una elevada velocidad o un acceso directo a los órganos del computador; por ejemplo, en los programas traductores o en programas de gestión del computador.

1.6 Ejercicios

1. Trate de describir con precisión mediante frases sencillas algunas tareas cotidianas, como leer un libro o realizar un trayecto en autobús.

2. Dado el algoritmo de Euclides que se menciona en el apartado 1.5.1,

$$\begin{array}{l}\text{Sean } A, B, R \in \mathbb{N}.\\ \text{Mientras } B \neq 0, \text{ hacer:}\\ \quad \left\{ \begin{array}{l} R \leftarrow A \textit{ modulo } B \\ A \leftarrow B \\ B \leftarrow R \end{array} \right.\\ \text{Escribir } A\end{array}$$

   trate de seguirlo, utilizando lápiz y papel (por ejemplo, para $A = 12$ y $B = 8$). Está formulado desde el punto de vista del procesador: cuando se dice "leer A, B", el procesador debe pedir dos valores para las variables A y B al usuario; la instrucción "mientras $b \neq 0$ hacer ..." significa que mientras la condición sea cierta hay que realizar las operaciones

comprendidas entre dicha instrucción (...) que, en nuestro caso, es una secuencia de tres. La operación "módulo" expresa el resto de la división entera, y el símbolo "←" indica que, tras calcular el resultado de la expresión a su derecha, se retendrá como el valor de la variable a su izquierda.

3. Clasifique los siguientes elementos del computador como pertenecientes al hardware, al software del sistema o al de aplicaciones: pantalla, teclado, unidad de entrada y salida, programa del sistema operativo, compilador, procesador de textos, etc.

1.7 Comentarios bibliográficos

Existe una gran cantidad de textos de introducción a la informática que se pueden recomendar con carácter general. Aun ciñéndonos a los escritos en castellano o traducidos, existen bastantes textos excelentes sobre el tema, con diversos enfoques.

En [GL86] encontramos un panorama general sobre informática en el que se utiliza como tema unificador el concepto de algoritmo, que sus autores juzgan como el concepto central de la computación. Se trata de un texto apropiado para el principiante y de lectura amena. El libro de Bishop ([Bis91]) es también un libro muy asequible de introducción en informática.

[PLT89] trata con gran detalle la estructura física de los computadores, por lo que interesará a quienes deseen detenerse en este aspecto de la informática.

Recomendamos [FSV87] a quien desee dirigirse hacia los fundamentos teóricos de la informática (la teoría de algoritmos, la lógica y la teoría de autómatas y lenguajes formales).

Por último, puesto que la informática es una ciencia reciente, cuenta con muchos vocablos nuevos, en su mayoría anglicismos, que muchas veces se emplean de forma ilegítima, o que se aplican con un sentido equivocado. En [VJ85] y [MA85] puede consultarse el significado de esos términos.

Capítulo 2

Representación digital de la información

La representación de la información en los computadores digitales persigue dos objetivos: en primer lugar, procesarla, permitiendo su manipulación eficiente, para lo cual se han ideado diferentes convenios, de los que veremos los más importantes; y en segundo, asegurarla contra errores durante su almacenamiento o durante las transmisiones, lo que se consigue incorporando en la codificación el empleo de la redundancia para detectar y corregir dichos errores.

2.1 Conceptos previos

2.1.1 Información analógica y digital

Las *magnitudes continuas* son las que pueden adoptar los infinitos valores de un intervalo de números reales, tales como la longitud de un segmento, velocidad, temperatura, intensidad de un sonido, etc.

Las *magnitudes discretas* tienen naturaleza discontinua, tales como la longitud (número de sílabas) de una palabra, capacidad (número de pasajeros) de un vehículo, etc.

En la práctica, es frecuente que las magnitudes continuas sean tratadas como discretas: el peso de una persona (que se redondea en kilos); la temperatura (en grados y décimas de grado); la longitud de un segmento, medida con un dispositivo de precisión hasta los milímetros.

En relación con ambos tipos de magnitud se considera la información *analógica*, que es de naturaleza continua, pudiendo tomar infinitos valores; y la información *digital*, que es de naturaleza discreta. Aunque esta última puede tomar infinitos valores ($\mathbb{N}$), en un computador digital la información es discreta y, además, finita.

En las calculadoras, la *digitalización* de variables analógicas produce un efecto de redondeo, que debe ser tenido en cuenta y tratado convenientemente para evitar errores de cálculo (véase la sección 2.3); en el monitor de un ordenador, supone el ajuste de la imagen proyectada sobre una matriz de puntos.

2.1.2 Unidades de información en los sistemas digitales

La razón de ser de un computador es el procesamiento de información. Para poder hablar con propiedad de este procesamiento, debemos definir unidades de medida que nos permitan cuantificar de algún modo la acción del computador sobre la información suministrada. Consideramos las siguientes:

- *Bit* (*BInary digiT*) es la cantidad de información que puede almacenarse en una variable binaria. No hay que confundir el bit con la variable ni con su valor: una *variable binaria* es la que puede tomar dos valores estables: 0 ó 1, blanco o negro, sí o no, etc.

  La necesidad de codificar informaciones más complejas ha llevado a agrupar varios bits, apareciendo así las siguientes unidades:

- El *byte* u *octeto* es la cantidad de información que puede codificarse en 8 bits; representa por tanto $2^8 = 256$ valores.

- La *palabra* se define en relación con la máquina considerada, como la cantidad de información que la máquina puede manejar de una sola vez. Para evitar equívocos, se habla de palabras de 8 bits, 16 bits, 32 bits, etc.
- *1 Kbyte* $= 2^{10}$ bytes $= 1.024$ bytes. Se suele llamar kilobyte, aunque esto puede resultar equívoco, ya que el prefijo "kilo" significa 1.000 (y no 1.024).
- *1 Mbyte* $= 1.048.576$ bytes ($2^{20} = 1.024^2$). Análogamente, debe advertirse que "mega" no significa un millón en este contexto.

2.1.3 Sistemas de numeración posicionales

Aunque se conocen sistemas no posicionales, tales como el de numeración romana o el sexagesimal, que usamos para medir el tiempo y los ángulos, el sistema de numeración más difundido en la actualidad es sin duda el sistema decimal posicional, o sistema arábigo-hindú, inventado hacia el siglo VIII.

Por otra parte, en el contexto de la informática se usan frecuentemente sistemas de numeración posicional en bases tales como 2 (ya que el bit tiene dos posiciones), 16 (como compactación de palabras de 4 bits), etc.

Para aprender a manejarlos, se recurre frecuentemente a la analogía con el sistema de numeración más conocido: el de base 10. Se llama *decimal* porque cada cifra o dígito puede tomar diez posibles valores: del 0 al 9; se llama *posicional* porque el valor real de cada dígito depende de su posición.

$$10475 = 1 * 10^4 + 0 * 10^3 + 4 * 10^2 + 7 * 10^1 + 5 * 10^0$$

A la cantidad 10 se le llama *base*; las potencias de 10 son los pesos asociados a cada posición, y los factores o coeficientes de cada peso son las *cifras* de la representación. También se podría haber representado en forma polinómica del siguiente modo:

$$10475 = 1 * 10^4 + 0 * 10^3 + 47 * 10^1 + 5 * 10^0$$

pero la primera forma es la única donde las cifras son todas menores que la base. En general, esta afirmación adopta la siguiente forma, cuya demostración se incluye al final de este capítulo.

Teorema 2.1 *En un sistema de numeración en base $b > 1$, todo entero N positivo tiene una única representación de la forma*

$$N = c_p b^p + c_{p-1} b^{p-1} + \ldots + c_1 b^1 + c_0 b^0$$

donde $0 \leq c_i < b$ para todo $i = 0, 1, \ldots, p$

Conversión entre sistemas

En primer lugar, la expresión decimal de un número de cifras $c_p \ldots c_0$ en base b se obtiene sencillamente sumando los valores reales correspondientes a los diferentes dígitos:

$$[c_p \ldots c_0]_{(b} = c_p * b^p + \ldots + c_0 * b^0$$

Por ejemplo, $275_{(8} = 2 * 8^2 + 7 * 8^1 + 5 * 8^0 = 189_{(10}$

En segundo lugar, representar el número $241_{(10}$ en el sistema de base 5, equivale a expresarlo en forma polinómica con las sucesivas potencias de esa base, siguiendo la idea de la demostración del teorema:

$$\begin{array}{rrrrrrrrr}
241 & | & 5 & & \Rightarrow & 241 & = & 48 * 5 + 1 \\
\cline{2-3}
1 & 48 & | & 5 & \Rightarrow & 48 & = & 9 * 5 + 3 \\
\cline{3-4}
 & 3 & 9 & | & 5 & \Rightarrow & 9 & = & 1 * 5 + 4 \\
\cline{4-5}
 & & 4 & 1 & \Rightarrow & 1 & = & 0 * 5 + 1
\end{array}$$

Por lo tanto,

$$\begin{array}{rcll}
241 & = & 48 & * 5 + 1 \\
 & = & (9 * 5 + 3) & * 5 + 1 \\
 & = & ((1 * 5 + 4) * 5 + 3) & * 5 + 1 \\
 & = & 1 * 5^3 + 4 * 5^2 + 3 * 5^1 + 1 * 5^0 & = 1431_{(5}
\end{array}$$

Sistemas de numeración más usuales

El sistema más empleado en electrónica digital es el de base 2, llamado binario (natural). En informática tienen interés los sistemas cuya base es una potencia de dos: 2, 4, 8, 16. La siguiente tabla recoge los primeros números naturales, expresados en algunos de esos sistemas y en el decimal:

dec.	binario	octal	hexad.
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8

dec.	binario	octal	hexad.
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11

En el sistema hexadecimal se usan los dígitos $0, \ldots, 9, A, \ldots F$ para las cantidades *cero, ..., nueve, diez, ..., quince* respectivamente. Así por ejemplo, $C7A_{(16} = 12 * 16^2 + 7 * 16^1 + 10 * 16^0 = 3194_{(10}$, ya que los valores de A y C en el sistema de base 16 son 10 y 12, respectivamente.

Se observa que, en una base cualquiera b, con N cifras (o menos) es posible expresar b^N cantidades distintas; inversamente, para poder componer C combinaciones distintas se necesita disponer de un número de cifras igual a $\log_b C$, redondeado por exceso.

Como consecuencia de lo anterior, cuanto mayor sea la base adoptada se pueden expresar más cantidades (combinaciones) para un número fijo de cifras; inversamente, cuanto mayor sea la base, es posible usar menos cifras para expresar una misma cantidad.

Observación

La conversión de binario en octal o en hexadecimal se puede abreviar del siguiente modo:

$$
\begin{array}{llllllll}
11\ 001\ 111\ 010\ 101\ 100_{(2} & & & & & & & \\
= & 11 & 001 & 111 & 010 & 101 & 100 & \\
= & 3 & 1 & 7 & 2 & 5 & 4 & = \quad 317254_{(8} \\
= & 1 & 1001 & 1110 & 1010 & 1100 & & \\
= & 1 & 9 & 14 & 10 & 12 & = & 19EAC_{(16}
\end{array}
$$

¿A qué se debe el funcionamiento de este mecanismo?

Operaciones aritméticas en base dos

Para las operaciones elementales se usan las tablas correspondientes a la base de que se trate. Por ejemplo, para el caso binario la tabla de sumar es la siguiente:

+	0	1
0	0	1
1	1	10

Y entonces, son válidas las reglas conocidas para las operaciones en base diez. Por ejemplo, en el sistema binario natural, tenemos:

$$
\begin{array}{rr}
 & 1001 \\
+ & 1011 \\
\hline
 & 10100
\end{array}
\qquad\qquad
\begin{array}{rr}
 & 100101 \\
- & 11011 \\
\hline
 & 01010
\end{array}
$$

Para la resta se usa frecuentemente el método del complemento: en lugar de la resta propuesta, se halla la suma correspondiente complementando el sustraendo ($min - sus \rightarrow min + comp(sus)$, siendo el complemento ($comp$) el número resultante de cambiar cada cero por un uno y viceversa), suprimiendo la cifra excedente, posiblemente aparecida por el arrastre, y sumando una unidad al resultado obtenido:

$$
\begin{array}{rr}
 & 100101 \\
- & 11011 \\
\hline
 &
\end{array}
\quad \rightarrow \quad
\begin{array}{rr}
 & 100101 \\
+ & 100100 \\
\hline
 & \underline{1}001001
\end{array}
\quad \rightarrow \quad 001010
$$

2.2 Representación digital de los datos

En los sistemas digitales, no resulta viable dar una representación válida para codificar todos los números; por otra parte, los diversos sistemas empleados dan diferentes tratamientos a números tan usuales como el uno (según se considere como real o como entero). Estudiaremos diversos convenios para diferentes conjuntos de números, así como sus limitaciones.

En este apartado, supondremos que disponemos de un espacio de N bits, con lo que es posible representar 2^N enteros distintos.

2.2.1 Representación de los números enteros

Números enteros positivos

Si se considera únicamente números enteros positivos, con N bits de espacio sería posible representar los números de 0 a 2^N-1. La forma más natural de lograrlo consiste en interpretar cada combinación mediante la cantidad que representa en binario. Por ejemplo, con 1 byte (es decir, $N = 8$) se representarían los números $0, \ldots, 255$ en este sistema.

Números enteros con signo. Convenio del signo-magnitud

Para representar los números enteros (con signo), el sistema más simple es el convenio de signo-magnitud, consistente en reservar el primer dígito binario para codificar el signo (suele representarse el signo $+$ con un cero y el $-$ con un uno) y los siguientes $N-1$ para el valor absoluto. Así, en este sistema tienen representación las cantidades $\pm 0, \ldots, \pm(2^N-1)$. En el caso particular de 1 byte, esas cantidades son $-127, \ldots, -0, +0, \ldots, +127$.

En este sistema, la aritmética es bastante simple: la suma de cantidades del mismo signo y la resta de cantidades de distinto signo siguen la regla básica en binario. Para sumar cantidades de distinto signo, o restar cantidades del mismo signo, resulta más práctico el método del complemento.

Se observa el inconveniente de que el cero tiene una doble representación, por lo que el test de la comparación para la igualdad en este sistema debe tenerlo en cuenta. Por otra parte, la suma/resta de signos y valores absolutos necesita dos algoritmos distintos, que no resultan muy eficientes. Los siguientes sistemas surgen precisamente para tratar de paliar estas deficiencias.

Números enteros con signo. Complemento restringido

Para comprender mejor el funcionamiento de este convenio en binario, conviene introducirlo primero en base diez. Si consideramos palabras de $N = 2$ dígitos (decimales), es posible representar $10^2 = 100$ cantidades distintas. En este convenio se opta por considerar los números negativos de $\{-49, \ldots, -0\}$, y los positivos de $\{0, \ldots, 49\}$, del siguiente modo: los números positivos se representan en decimal natural, mientras que para cada negativo $-z$ se toma la cantidad $99 - z$ (complemento de z respecto de $10^2 - 1$, que es la N=2 potencia de la base menos uno). Así por ejemplo, tenemos:

$$repr(29) = \boxed{2}\boxed{9}$$

$$repr(-29) = 99 - 29 = \boxed{7}\boxed{0}$$

Con este convenio, las cantidades

$$-49, -48, \ldots, -0, 0, 1, \ldots, 48, 49$$

se representan respectivamente mediante

$$\boxed{5}\boxed{0}\,,\ \boxed{5}\boxed{1}\,,\ \ldots\,,\ \boxed{9}\boxed{9}\,,\ \boxed{0}\boxed{0}\,,\ \boxed{0}\boxed{1}\,,\ \ldots\,,\ \boxed{4}\boxed{8}\,,\ \boxed{4}\boxed{9}$$

Aunque se observa el inconveniente de que el cero tiene dos representaciones, la ventaja de este convenio consiste en que la suma de dos números, sea cual fuere su signo, se lleva a cabo con un mismo algoritmo, que sólo se diferencia de la suma de enteros en que el posible arrastre se agrega al resultado final:

$$\begin{array}{rr} & +\,23 \\ +\; & -\,15 \\ \hline & \end{array} \quad \rightarrow \quad \begin{array}{rr} & \boxed{2}\boxed{3} \\ + & \boxed{8}\boxed{4} \\ \hline 1 & \boxed{0}\boxed{7} \end{array} \quad \overset{+1}{\rightarrow} \quad \boxed{0}\boxed{8}$$

Consideremos ahora esta representación con palabras de $N = 4$ dígitos binarios: las cantidades representables son

$$-7, -6, \ldots, -0, +0, \ldots, +6, +7$$
$$\rightarrow -111_{(2}, -110_{(2}, \ldots, -0_{(2}, +0_{(2}, \ldots, +110_{(2}, +111_{(2}$$

que, al complementar las negativas respecto de 1111, resulta:

$$\rightarrow 1000, 1001, \ldots, 1111, 0000, \ldots, 0110, 0111$$

Números enteros con signo. Complemento auténtico

Empezamos nuevamente con la base decimal como punto de partida, y consideremos también palabras de longitud $N = 2$. Ahora se opta por considerar los números negativos de $\{-50, \ldots, -1\}$, y los positivos de $\{0, \ldots, 49\}$, del siguiente modo: los números positivos se representan en binario natural, mientras que para cada negativo $-z$ se toma la cantidad $100 - z$ (complemento de z respecto de 10, que es la base para $N = 2$).

$$repr(29) = \boxed{2}\boxed{9}$$

$$repr(-29) = 100 - 29 = \boxed{7}\boxed{1}$$

Con frecuencia se emplea otra regla equivalente para complementar los números negativos, consistente en añadir una unidad al correspondiente complemento restringido.

Con este convenio, las cantidades

$$-50, -49, \ldots, -1, 0, 1, \ldots, 48, 49$$

se representan respectivamente mediante

$$\boxed{5}\boxed{0}, \boxed{5}\boxed{1}, \ldots, \boxed{9}\boxed{9}, \boxed{0}\boxed{0}, \boxed{0}\boxed{1}, \ldots, \boxed{4}\boxed{8}, \boxed{4}\boxed{9}$$

Este sistema tiene las mismas ventajas que el anterior, y además el cero se representa de un único modo.

Como en el caso anterior, consideremos ahora la base dos, con palabras de $N = 4$ dígitos, donde tienen cabida las cantidades

$$-8, -7, \ldots, -1, +0, \ldots, +7$$
$$\rightarrow -1000_{(2}, -111_{(2}, \ldots, -1_{(2}, +0_{(2}, \ldots, +110_{(2}, +111_{(2}$$

que, al complementar las negativas respecto de 10000, resulta:

$$\rightarrow 1000, 1001, \ldots, 1111, 0000, \ldots, 0110, 0111.$$

En este sistema, la suma también se lleva a cabo con un mismo algoritmo, igual al usado para el complemento restringido, pero ignorando la posible cifra de arrastre:

$$\begin{array}{rr} & +\ 23 \\ +\ & -\ 15 \\ \hline & \end{array} \quad \rightarrow \quad \begin{array}{rcc} & 2 & 3 \\ + & 8 & 5 \\ \hline 1 & 0 & 8 \end{array} \quad \rightarrow \quad \begin{array}{|c|c|}\hline 0 & 8 \\ \hline \end{array}$$

Funcionamiento de las operaciones aritméticas

Debe subrayarse que las operaciones anteriores no coinciden con las aritméticas, debido a la posibilidad de que se produzca un desbordamiento; los diferentes sistemas responden ante esta circunstancia de diferentes modos: por ejemplo, ignorando la última cifra de arrastre, o interrumpiendo su trabajo para delatar una condición de error. Por lo tanto, es necesario prever esta posibilidad y conocer de qué modo reacciona nuestro sistema. Sobre este asunto volveremos más adelante, dentro de este mismo capítulo.

Formatos de los números enteros en las computadoras

Entre los convenios presentados, el más frecuente es el del *complemento auténtico en base dos*, llamado simplemente *complemento a dos*. Ahora bien, dependerá de la longitud de palabra la cantidad de combinaciones posibles y, por tanto, el rango de enteros considerado. Por otra parte, aunque ciertos sistemas trabajan con palabras de longitud variable, lo corriente es optar por uno o varios formatos con tamaño fijo: simple (1 byte), doble (2 bytes), cuádruple (4 bytes) u óctuple (8 bytes).

2.2.2 Representación de los números reales

Debe resaltarse que, en general, sólo resulta posible representar aproximaciones de los números reales mediante números decimales, con sólo unas pocas cifras significativas. En el siguiente apartado estudiaremos los efectos de este redondeo.

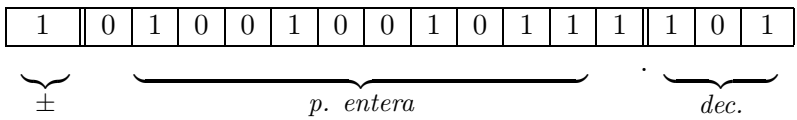
En este apartado, supondremos que disponemos de un espacio de N bits, con lo que es posible representar 2^N enteros distintos.

Convenio con coma fija

Si disponemos de un espacio de N bits para representar un número real, la característica principal de este convenio es la reserva implícita de algunos bits fijos para la parte decimal, asumiéndose la coma en una posición fija. A su vez, existen los siguientes modos de representación en coma fija:

- Sistema signo y valor absoluto

  En este convenio se reserva un bit para codificar el signo, y del resto se destina una cantidad fija para representar el valor absoluto de la parte entera, y los demás para la decimal:



  Si consideramos por ejemplo $N = 16$, siendo el primer bit el que codifica el signo, los siguientes 12 los de la parte entera, y los 3 restantes los de la parte decimal, resulta que la representación anterior significaría

$$\begin{aligned} &-010010010111.101 \\ &\quad = -(2^{10} + 2^7 + 2^4 + 2^2 + 2^1 + 2^0 + 2^{-1} + 2^{-3}) \\ &\quad = -1175'625. \end{aligned}$$

- Complemento restringido y complemento auténtico

  Sea $D = 3$ el número (fijo) de decimales asumidos. La representación de un número x, en cualquiera de estos convenios, consiste en codificar en su lugar la parte entera de $x * 2^3$ en el convenio elegido. La cantidad despreciada al truncar los decimales de $x * 2^3$ es menor que $0'125_{(10} (= 2^{-3})$.

Por ejemplo, para representar $-2'8$ en palabras de $N = 8$ y $D = 3$, debemos conformarnos con una aproximación: al ser $D = 3$, sólo podemos representar cantidades decimales múltiplos enteros de $0'001_{(2} = 2^{-3}_{(10} = 0'125_{(10}$. Como $trunc(-2'8 * 8) = trunc(-22'4) = -22$, representamos $-22/8 = -2'75$. Comprobamos que esta cantidad es efectivamente múltiplo de $0'125$, y que el siguiente múltiplo ($-2'875$) excede la cantidad inicial ($-2'8$).

- Signo y valor absoluto:

$$2.75 = 2 + \frac{1}{2} + \frac{1}{4} \quad = \quad \underbrace{\boxed{1}}_{\pm} \underbrace{\boxed{0}\boxed{0}\boxed{1}\boxed{0}}_{p.\ entera} . \underbrace{\boxed{1}\boxed{1}\boxed{0}}_{dec.}$$

- Complemento restringido (8 dígitos):

$$-22 = -\left\{ \begin{array}{r} 11111111 \\ 10110_{(2} \end{array} \right\} \rightarrow 11101001 = \boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{1}$$

- Complemento auténtico (8 dígitos):

$$-22 = -\left\{ \begin{array}{r} 100000000 \\ 10110_{(2} \end{array} \right\} \rightarrow 11101010 = \boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{0}$$

Convenios con coma flotante

El principal inconveniente de la coma fija consiste en condicionar el orden de magnitud de las cantidades codificadas. Un sistema más general deberá adaptarse a órdenes tan distantes como los de la masa de

la tierra y la del átomo. El sistema más conocido con esta característica es la llamada notación exponencial (o científica):

$$\begin{array}{lcl} 140 = +0.14 * 10^3 & = & +0.14E3 \\ 6.02215 * 10^{23} & = & +0.602215E24 \\ 0.00000015 & = & +0.15E-6 \end{array}$$

Los convenios con coma flotante funcionan del mismo modo, dividiendo la información sobre una cantidad en tres partes: su signo, su mantisa (es decir, las cifras significativas de mayor orden), y el exponente (que expresa el orden de magnitud). Así por ejemplo, consideremos la siguiente representación en palabras de N bits:

- El signo, adscrito al primer bit, donde el uno representa al signo menos.

- El exponente (z), situado en los siguientes e bits, puede tomar los valores de $\{-2^{e-1}, \ldots, 2^{e-1} - 1\}$. Frecuentemente se representa desplazado en $+2^{e-1}$ unidades:

$$z \in \{-2^{e-1}, \ldots, 2^{e-1} - 1\} \Rightarrow repr(z) \in \{0, \ldots, 2^e - 1\}$$

- Para la mantisa quedan $M = N - e - 1$ bits con lo que, para su representación, se toman los primeros M dígitos de su escritura en binario natural.

Como ejemplo concreto, consideremos el convenio estándar IEEE 754 (real con precisión simple, o real corto), que es una de las codificaciones de reales más utilizadas. Su representación requiere un total de 32 bits, de los cuales un bit se utiliza para el signo, 8 bits para el exponente y 23 bits para la mantisa. Este convenio en concreto, utiliza un truco llamado *bit oculto*: como el bit más significativo de la mantisa es un 1, ahorramos un bit simplemente asumiéndolo. Se logra así espacio para una mantisa de 24 dígitos significativos. El inconveniente de emplear un bit oculto consiste en que se requiere una representación especial para el cero.

2.2.3 Limitaciones de los sistemas de representación digital de los números

Debido a que los sistemas de codificación considerados están inmersos en sistemas finitos, sólo pueden representar una cantidad finita de elementos distintos. Esta limitación resulta crítica a la hora de representar elementos pertenecientes a conjuntos infinitos, como son los de los números enteros o reales. Como consecuencia, pueden producirse situaciones de error no deseables, por lo cual se hace necesario estudiar el alcance de esas limitaciones, así como la manera de afrontarlas.

Limitaciones en los enteros

En los enteros, las representaciones se limitan a un intervalo reducido $[mín,\ máx]$, de cardinal no superior a 2^n, siendo n el tamaño (en bits) de la representación. Así, cuando surge la codificación de números fuera del rango considerado, se produce el llamado desbordamiento (*overflow* en inglés).

Por ejemplo, en una representación con 4 bits serán posibles 16 configuraciones. Si optamos por el convenio de complementación auténtica, el intervalo considerado será $[-8, 7]$. Veamos qué ocurre al sumar 5 y 6.

$$\begin{array}{rcrcrcl} & 5 & \rightarrow & & 0101 & & \\ + & 6 & \rightarrow & + & 0110 & & \\ \hline & & & & 1011 & \rightarrow & -5 \end{array}$$

El comportamiento de las operaciones de suma y resta en el sistema de complementación auténtica para una representación de n bits se puede describir así: llamemos x' e y' a la representación de x e y, y $+'$ y $-'$ a las operaciones de suma y resta en este sistema, $\forall x, y \in [-2^{n-1}, 2^{n-1} - 1]$, tenemos:

$$x' \pm' y' = \begin{cases} (x \pm y + 2^n)', & \text{si } x \pm y < -2^{n-1} \\ (x \pm y)', & \text{si } x \pm y \in [-2^{n-1}, 2^{n-1} - 1] \\ (x \pm y - 2^n)', & \text{si } x \pm y > 2^{n-1} - 1 \end{cases}$$

Para paliar en cierta medida esta limitación, es frecuente incorporar tipos de datos correspondientes a rangos de enteros más amplios

(por ejemplo, duplicando la longitud de palabra de la representación), pudiendo así manipular con seguridad enteros de mayor orden. Así por ejemplo, el cálculo de 8! desbordará un sistema de complemento auténtico con 16 dígitos, resultando en cambio correcto para palabras más largas. En ocasiones, el programador tiene la posibilidad de cambiar de estrategia para evitar el desbordamiento. Por ejemplo, el cálculo de $\binom{8}{3}$ se puede obtener evaluando $\frac{8!}{3!*5!}$ o $\frac{8*7*6}{3*2*1}$ indistintamente; sin embargo, el primero de ellos necesita manipular números mayores, por lo que el segundo nos permite eludir en cierta medida el error por desbordamiento.

Por otra parte, muchos de los lenguajes que se comercializan en la actualidad ofrecen la opción de verificar o no los desbordamientos que puedan producirse durante la ejecución, para que los consiguientes errores no pasen inadvertidos.

Finalmente, la mayoría de los lenguajes de alto nivel presentan sistemas de construcción de tipos de datos con capacidad para definir codificaciones (y operaciones de manipulación sobre los objetos cifrados) a la medida de nuestras necesidades.

Limitaciones en los reales

En este caso, incluso limitándonos a los reales contenidos en un pequeño intervalo acotado, éstos serían infinitos, por lo cual las limitaciones en su representación no sólo afectan al tamaño de los números considerados, sino también a la precisión. En efecto, en cualquiera de los convenios estudiados se toma como mantisa sólo cierto número reducido de dígitos(binarios) significativos, despreciándose los demás.

En otras palabras, cada codificación de un número real en coma flotante representa en realidad un intervalo, cuyo tamaño varía según la magnitud del real representante. Por consiguiente, la distribución de representantes no es uniforme: un intervalo tan pequeño como [0'1, 1] cuenta con tantos representantes como [1000, 10000].

Además de las limitaciones de la precisión, la representación de los reales está limitada por su tamaño: existe un valor a partir del cual no

hay representantes reales y que determina el rango de desbordamiento.

Debe observarse en este punto que existen cantidades cuya expresión decimal es exacta, sin serlo su expresión escrita en binario natural (por ejemplo, $0'1_{(10} = 0'0\,0011\,0011\,\ldots_{(2}$), por lo que, en principio, debe desconfiarse de la precisión en la codificación de todo número no entero.

Aunque esta diferencia entre una cantidad y su representación es en muchos casos despreciable, su aparición puede ocasionar grandes desviaciones respecto del comportamiento teórico.

Una situación así se produce cuando se comparan dos números reales para determinar su igualdad, ya que el resultado de la comparación ignora si esa diferencia producida es pequeña o grande, lo que puede ocasionar una respuesta drásticamente distinta de la correcta. Así por ejemplo, la ejecución del siguiente programa escrito en Pascal

```
Program errores (output);
    var suma: real;
    begin
        suma := 0;
        repeat
            suma := suma + 0.1;
            writeln(suma);
        until suma = 1
    end.
```

no para.[1] Frecuentemente esta situación puede evitarse cambiando la expresión $x = y$ por $\mid x - y \mid < \varepsilon$, siendo ε la diferencia admitida.

En realidad, la aparición de errores intolerables es posible incluso en expresiones sencillas. Concretamente, deberían evitarse las operaciones de suma y resta, cuando uno de los operandos es muy pequeño en comparación con el otro, y la división cuando el divisor es cero o "próximo a cero". Por ejemplo, la relación $|t/x| \leq \varepsilon$ debería sustituirse por $|t| \leq \varepsilon * |x|$.

[1] Suponiendo una representación en binario puro.

Más aún, aun cuando se trate de un error despreciable, es frecuente que ese error intervenga en cálculos repetidos, como ocurre en la manipulación de matrices, el cálculo con series recurrentes, etc.; en tales situaciones, la propagación de un error inicial, aunque pequeño, puede generar un error mucho mayor. El estudio y la cuantificación de los errores, así como su propagación en cálculos repetitivos y los métodos para encontrar soluciones satisfactorias, escapan del alcance y objetivos de este curso, siendo materia propia de los métodos numéricos.

Otros sistemas de representación (paquetes matemáticos)

Durante la pasada década, se han desarrollado y difundido paquetes de programas matemáticos, capaces de resolver eficientemente un extenso número de problemas, manipulando expresiones tanto numéricas como simbólicas. Para ello están provistos (aparte de otros mecanismos) de potentes sistemas aritméticos de representación no convencionales. Veamos dos ejemplos de estas posibilidades:

- Capacidad de representar enteros con tamaño limitado sólo por la memoria del ordenador, fracciones y reales con una precisión arbitraria, elegida por el usuario. Por ejemplo:

  sea precisión_decimales = 25
  escribir pi

  3.14159265358979323846264338

- Capacidad de establecer valores numéricos (reales o complejos) mediante su definición, en lugar de su cálculo, con lo que no hay pérdida de precisión. Así, es posible establecer sentencias del estilo de las siguientes:

  sea $x_0 := x$ **tal_que** $x^2 + 5 = 0$
  escribir $(1 - {x_0}^2)/2$

  cuya ejecución produciría la escritura de 3, exactamente.

2.2.4 Representación de los caracteres

Existen otros convenios, además de los numéricos, para representar los caracteres disponibles habitualmente en un teclado de computador: los dígitos, las letras minúsculas y mayúsculas, los signos de puntuación y de operación y otros símbolos especiales, tales como #, &, @, %, etc.

Inicialmente surgieron numerosos convenios para codificar los caracteres, variando el número n de bits empleados (y con él su capacidad de representación), así como la posición, entre 0 y $2^n - 1$, asignada a cada carácter. Sin embargo, en seguida se observó la necesidad de adoptar convenios normalizados, así como la conveniencia de que éstos tengan ciertas cualidades:

- Debe incluirse el juego de letras mínimo internacional, en dos intervalos de posiciones consecutivas, correspondientes a las letras minúsculas y mayúsculas.

- Los caracteres correspondientes a los dígitos deben ocupar también posiciones correlativas, de "fácil" cifrado y descifrado.

Además, cuando la capacidad de representación lo permita, será deseable que en un sistema sea posible:

- Añadir otros caracteres más específicos: frecuentemente, los propios de una lengua (las "á" y "ñ" españolas, la "û" francesa, etc.)

- Destinar ese exceso de capacidad a prevenir y subsanar posibles errores, como veremos en el siguiente apartado.

Uno de los convenios más extendidos en la actualidad es el ASCII, con 7 bits, por lo que admite hasta $2^7 = 128$ caracteres. Damos la siguiente tabla, omitiendo los primeros 32 caracteres, por ser caracteres de control.

32		44	,	56	8	68	D	80	P	92	\	104	h	116	t
33	!	45	-	57	9	69	E	81	Q	93	]	105	i	117	u
34	”	46	.	58	:	70	F	82	R	94	ˆ	106	j	118	v
35	#	47	/	59	;	71	G	83	S	95	_	107	k	119	w
36	$	48	0	60	¡	72	H	84	T	96	‘	108	l	120	x
37	%	49	1	61	=	73	I	85	U	97	a	109	m	121	y
38	&	50	2	62	¿	74	J	86	V	98	b	110	n	122	z
39	’	51	3	63	?	75	K	87	W	99	c	111	o	123	{
40	(	52	4	64	@	76	L	88	X	100	d	112	p	124	|
41	)	53	5	65	A	77	M	89	Y	101	e	113	q	125	}
42	*	54	6	66	B	78	N	90	Z	102	f	114	r	126	˜
43	+	55	7	67	C	79	O	91	[	103	g	115	s		

En este convenio, se observa en primer lugar que los dígitos decimales “0”, ..., “9” ocupan las posiciones 48, ..., 57: sus valores (0000, ..., 1001) coinciden con las terminaciones de sus posiciones (0110000, ..., 0111001).

Siguiendo la segunda condición, las letras mayúsculas y minúsculas se hallan situadas en las posiciones 65 a 90 y 97 a 122, respectivamente.

En la práctica no se utilizan palabras de 7 bits, siendo frecuente en cambio adoptar el byte (= 8 bits) como unidad. Así, es posible extender el convenio anterior y dar cabida a otros caracteres de uso también interesante en ciertas aplicaciones; he aquí algunos ejemplos:

Posición:	130	145	156	164	165	168	248
Carácter:	é	æ	£	ñ	Ñ	¿	°

2.2.5 Organización de datos más complejos

Los computadores no sólo almacenan y manipulan números y caracteres, sino que también deben organizar y tratar informaciones más complejas, tales como sucesiones de datos (por ejemplo, cadenas de caracteres), vectores, tablas, etc., ya sea formadas por datos simples o bien por conjuntos de información con alguna organización. Con tal finalidad se han ideado diversas estrategias; aunque su estudio excede el alcance de este capítulo, veamos un ejemplo orientativo.

Supongamos una máquina de 1 byte (tamaño de palabra). Si se adopta para los números enteros un convenio de 2 bytes, cada número ocupará dos palabras consecutivas. Una forma natural y sencilla de organizar un vector de n enteros consiste en situarlos secuencialmente, empezando en las posiciones $m_0, m_0 + 2, \ldots, m_0 + 2(n-1)$, y la componente i-ésima del vector reside en las posiciones $m_0+2(i-1)$ y m_0+2i-1, para $i \in \{1, \ldots, n\}$.

Si se tratase de una matriz de $m * n$, cuyas componentes ocupan k palabras de memoria, se establece fácilmente la posición inicial para la componente i, j-ésima:

$$m_0 + k(n(i-1) + j - 1)$$

para $i \in \{1, \ldots, n\}$, $j \in \{1, \ldots, m\}$ y siendo m_0 la posición inicial de la matriz.

2.2.6 Representación de las instrucciones

Aunque podría parecer impropio considerar las instrucciones como informaciones, lo cierto es que los programas se almacenan en la memoria del ordenador y manipulan del mismo modo que los datos. De hecho, desde el punto de vista de la máquina, un programa no es más que una secuencia de elementos, cada uno de los cuales es un descriptor de una instrucción elemental.

El formato de las instrucciones (es decir, de sus descriptores) depende de la máquina destinada a seguirlas. Si consideramos por ejemplo una máquina *de tres direcciones* (véase cap. 3), sus instrucciones se componen de cuatro campos que codifican respectivamente la operación que debe efectuarse y las posiciones de memoria en que se encuentran los (como máximo dos) argumentos y la de la palabra donde debe situarse finalmente el resultado. En el capítulo siguiente se presenta con detalle uno de estos formatos.

2.3 Códigos redundantes

2.3.1 Información y redundancia

Información e incertidumbre.- El término información tiene en general un significado muy amplio: piénsese en la información proporcionada por una fotografía o un poema. Por eso, en el contexto de la informática como ciencia que trata sobre el tratamiento automático de la información, se asocia a ésta un significado más restringido y manejable.

Para definir este significado, consideremos que deseamos determinar un cierto fenómeno, que puede presentar una cantidad finita de estados. Entonces, una información (sobre ese fenómeno) es una sentencia capaz de aportar algún conocimiento sobre tal fenómeno; esto es, capaz de delimitar en cierta medida su estado.

Así por ejemplo, si consideramos el fenómeno "colores del atuendo que llevaba anoche el asesino del callejón", sabiendo que el pantalón era negro o marrón, que la camisa era azul, gris o marrón, y que el sombrero era gris o negro, resulta que el número de estados es 12. Una información sobre este fenómeno podría consistir en la sentencia "la camisa y el pantalón eran de distinto color"; con esta información, sólo son posibles 10 estados de nuestro fenómeno. Ahora, la afirmación "la camisa y el pantalón no eran ambos marrones" resulta redundante puesto que, considerándola, siguen siendo posibles los mismos 10 estados que ignorándola.

En lugar de hablarse de cantidad de información, resulta más fácil manipular la de *incertidumbre*. Una medida indirecta de ello consiste en la cantidad de estados posibles: su grado de indeterminación. Si cierto fenómeno ofrece 12 estados posibles, su grado de indeterminación es 12, pero resulta más conveniente considerar como medida de la incertidumbre el logaritmo (en base dos) de esta cantidad, puesto que la incertidumbre crece de forma exponencial (con base dos) respecto de la longitud (número de bits) del mensaje. Esta medida de la incertidumbre se llama *entropía* asociada a un fenómeno, y se denota mediante H:

$$H = \log_2(\textit{número de estados posibles de un fenómeno})$$

Entonces, la información asociada a una sentencia se define como disminución de incertidumbre proporcionada. Siendo $H_0 = \log_2(n_0)$ y $H_1 = \log_2(n_1)$ las entropías correspondientes a los estados previo y posterior a la sentencia, con n_0 y n_1 posibles estados, respectivamente, tenemos:

$$I = H_0 - H_1 = \log_2(n_0) - \log_2(n_1) = \log_2\left(\frac{n_0}{n_1}\right).$$

Por ejemplo, para la primera sentencia, tenemos:

$$I_1 = \log_2 1'2 > 0$$

mientras que la segunda proporciona una cantidad de información nula, al ser redundante.

La redundancia en la codificación.- En los convenios de codificación/decodificación estudiados hasta ahora, se ha supuesto que su transmisión o su almacenamiento se efectúa siempre sin ruido. En ellos, el objetivo principal consiste en diseñar convenios eficaces (sin emplear más dígitos que los estrictamente necesarios, minimizando así la redundancia), unívocos (donde el cifrado es único) y sin ambigüedad, siendo única toda decodificación.

Para prever la posibilidad de que se produzcan perturbaciones, se han ideado métodos capaces de descubrir en ciertas condiciones cuándo se ha alterado un mensaje (*códigos detectores*), así como otros capaces de restituir su estado inicial (*códigos correctores*).

Ambas clases de mecanismos se apoyan en el uso de la redundancia. Anticipamos un par de ejemplos para aclarar ambos tipos de mecanismos.

Adición de un bit de paridad.- Para un mensaje de n bits se añade uno, cuyo valor consigue que haya en total un número par de unos. Se ignora la posibilidad de que se altere más de un bit, por considerarla extremadamente improbable. En la interpretación del mensaje se verifica previamente la paridad, detectándose un error si se ha producido, aunque no será posible identificar cuál para restablecer su estado.

Código dos entre tres.- Este mecanismo consiste sencillamente en triplicar las copias de cierta información. Se consideran dos posibilidades: que no se produzca ninguna alteración, o que se produzca en una de las copias, descartándose mayores perturbaciones. En ambos casos, se interpreta el mensaje cifrado en la mayoría de las copias: dos (al menos) entre tres.

2.3.2 Códigos sólo autodetectores: *p de n*

Si en una palabra de n bits (que admite 2^n configuraciones) establecemos la restricción de considerar válidas sólo aquéllas con exactamente p unos (y $n-p$ ceros), será posible detectar si se efectúa una perturbación simple, o una múltiple, siempre que no se alteren tantos unos como ceros. En un código $p\,de\,n$, el grado de indeterminación es el número de permutaciones con repetición de p unos y $n-p$ ceros

$$\binom{n}{p} = \frac{n!}{p!(n-p)!}$$

que alcanza su máximo cuando $p = n\,div\,2$.

2.3.3 Códigos autocorrectores: Hamming

Fundamento

Consideremos un mensaje de n bits. La idea básica consiste en añadir un cierto número p de bits, cada uno de los cuales asegura un cierto subconjunto de los $n+p$ dígitos totales mediante un control de paridad. Se considera la posibilidad de que exista, a lo sumo, una alteración en uno de los $n+p$ bits y deseamos conocer si ha habido o no perturbación y, en caso afirmativo, cuál de los $n+p$ bits ha sido alterado. Puesto que el mensaje debe codificarse en los n bits, la información sobre el estado de perturbación debe cifrarse en los p bits, para poder así restablecerla. Por tanto, conocido n, p debe ser la mínima cantidad de bits tal que los 2^p estados posibles de los p bits de paridad acepten al menos $n+p+1$ estados distintos: no alteración (1) o la posición del bit alterado $(n+p)$. Es decir:

$$p = mín\, k \in \mathbb{N} \text{ tal que } 2^k \geq n + k + 1$$

Por otra parte, los p conjuntos de bits deben elegirse de modo que el estado de los p bits de paridad asociados a esos conjuntos permita localizar el bit alterado, en su caso y, si es posible, con facilidad.

Un modo de conseguir el objetivo descrito consiste en intercalar el bit i-ésimo en la posición 2^{i-1}, para $i = 1, \ldots, p$, siendo su conjunto asociado el de los dígitos cuyo número de posición, escrito en binario natural, tiene un 1 como cifra i-ésima.

Sea por ejemplo $n = 4$. Necesitamos $p = 3$ bits de paridad, que colocaremos en las posiciones 1, 2 y 4:

posición :	1	2	3	4	5	6	7
id. en binario :	001	010	011	100	101	110	111
	☐	☐	☐	☐	☐	☐	☐

Ahora, el bit $001_{(2}$ está asociado al conjunto de los bits del mensaje cuya posición acabe en 1 (1, 3, 5 y 7); el bit $010_{(2}$ está asociado al conjunto de los bits del mensaje cuya posición tenga un 1 en la segunda cifra (2, 3, 6 y 7), y el bit $100_{(2}$ está asociado al conjunto de los bits del mensaje cuya cifra inicial sea un 1 (4, 5, 6 y 7).

Codificación

Sencillamente, se trata de ajustar los bits de paridad con respecto a sus conjuntos asociados. Por ejemplo, si se desea transmitir el mensaje 0110

		0		1	1	0

bastará con ajustar el bit 1^0 con los 3^0, 5^0 y 7^0 (resultando un 1); el bit 2^0 con los 3^0, 6^0 y 7^0 (resultando un 1), y el bit 4^0 con los 5^0, 6^0 y 7^0 (resultando un 0):

1	1	0	0	1	1	0

Decodificación con autocorrección

Para rectificar y descifrar un mensaje recibido, se detectan en primer lugar los bits de paridad que reflejan alguna alteración. Si no hay ninguno, el mensaje se ha mantenido intacto durante la transmisión; en caso contrario, la suma de las posiciones de los bits alterados señala el bit modificado.

Por ejemplo, si el mensaje anterior se recibe así:

1	1	0	0	0	1	0

el control de paridad arroja el siguiente resultado:

$$\begin{aligned} \text{bit } 1^0(+3+5+7) &= 1\,(\text{impar}) \Rightarrow \text{alterado} \\ \text{bit } 2^0(+3+6+7) &= 2\,(\text{par}) \Rightarrow \text{sin alterar} \\ \text{bit } 4^0(+5+6+7) &= 1\,(\text{impar}) \Rightarrow \text{alterado} \end{aligned}$$

Al ser $1 + 4 = 5$, concluimos que el 5^0 bit es erróneo, por lo que el mensaje original era

1	1	0	0	1	1	0

Anexo: demostración del teorema 2.1

Incluimos aquí esta demostración por ser constructiva, mostrando el proceso de expresar una cantidad en cualquier base mayor que uno. Procederemos en cuatro fases: en primer lugar, la *existencia* de esa representación en las condiciones del teorema se demuestra fácilmente por inducción, dando un método para hallar esa construcción. En segundo, se demuestra (también por inducción) que ese método *converge* y a continuación que la solución que proporciona *equivale* efectivamente a la cantidad dada. Finalmente, se demuestra que no hay más que *una* representación de un número en una base (≥ 2) dada.

1. Dada la cantidad entera y positiva N, y la base $b > 1$, la representación de N en base b sigue el siguiente esquema:

$$repr_b(N) = \begin{cases} N & \text{si } N < b \\ repr_b(D).R & \text{en otro caso, donde} \\ & D = N \ div \ b \text{ y } R = N \ mod \ b \end{cases}$$

donde el punto expresa la separación entre las cifras de N, expresado en la base b, y donde *div* y *mod* representan, respectivamente, el cociente y el resto de la división entera.

2. La *convergencia* del método resulta obvia considerando que, partiendo de cualquier número entero positivo N, la secuencia $N = N_0, N_1, ...$ conduce al conjunto $\{0, ..., b\}$ mediante la aplicación de $N_i = N_{i-1} \ div \ b$, en un número finito, $trunc(\log_b N)$, de pasos.

3. La *equivalencia* queda demostrada por inducción sobre el número de cifras de la representación obtenida: cuando $N < b$, tenemos el caso base $repr_b(N) = N$; en caso contrario, basta considerar que la cantidad representada por $repr_b(D).R$ es

$$b * repr_b(D) + R = b * repr_b(N \ div \ b) + (N \ mod \ b)$$

Asumiendo ahora como hipótesis inductiva que $repr_b(N \ div \ b)$ representa la cantidad $N \ div \ b$, podemos expresar la cantidad anterior como

$$= b * (N \ div \ b) + (N \ mod \ b)$$

que es precisamente N: recuérdese que

$$dividendo = divisor * cociente + resto$$

4. En las condiciones del teorema, esa representación es *única*. Lo demostraremos por reducción al absurdo. Supongamos que es posible expresar una cantidad x de dos modos distintos, cuyas representaciones tienen las cifras $...a_0$ y $...a'_0$, siendo L la posición de las cifras distintas de mayor peso, y consideremos por ejemplo que es $a_L > a'_L$. Entonces, se tiene:

(a) La diferencia entre a_L y a'_L representa al menos b^L unidades:

$$a_L * b^L > a'_L * b^L \Rightarrow a_L * b^L \geq (a'_L + 1) * b^L = a'_L * b^L + b^L$$

(b) Por otra parte, la cantidad máxima que pueden representar los dígitos siguientes es de

$$\sum_{i=0}^{L-1} (b-1) * b^i = b^L - 1.$$

Resulta entonces que la diferencia que supone la cifra distinta de mayor orden ($\geq b^L$) no puede compensarse por ninguna combinación de las siguientes ($\leq b^L - 1$). Por tanto, si dos representaciones son distintas en alguna cifra, también lo son las cantidades representadas.

2.4 Ejercicios

1. Exprese los siguientes números en las demás bases:

binario puro	decimal	hexadecimal
1001110011		
	6723	
		1A9E

2. Efectúe las siguientes operaciones, en la base indicada:

   - $11011101_{(2} + 11110000_{(2}$
   - $11000101_{(2} * 101001_{(2}$
   - $A2396_{(16} + 24BC2_{(16}$
   - $A2396_{(16} * 1A_{(16}$

   Para el último apartado es recomendable construir previamente la tabla de multiplicar por A, en base hexadecimal.

3. Halle el rango de los posibles enteros,

   - si consideramos palabras de tamaño medio
   - si consideramos palabras de tamaño simple

- si consideramos palabras de tamaño doble

4. Represente 7 y -3 en los siguientes formatos, y realizar la suma correspondiente:
   - complemento restringido, en decimal
   - ídem, en binario
   - complemento auténtico, en decimal
   - ídem, en binario

   Haga lo mismo con 12 y 7.

5. Represente los números 35, 0'25 y 2'6 en los siguientes formatos:
   - en coma fija, con 5 bits enteros y 3 decimales
   - coma flotante: signo (1 bit), mantisa (9) y exponente (6)

6. Generalice la representación en memoria de vectores y matrices a matrices tridimensionales.

7. Se desea diseñar un código capaz de cifrar una información con 33 posibles estados.
   - ¿Cuántos bits son necesarios?
   - Con ese número de dígitos, ¿cuántos estados son posibles?
   - Un control del tipo $2\,de\,n$, ¿cuántos dígitos necesita?

8. Se desea transmitir mensajes de 31 bits, asegurándolos mediante un código de Hamming.
   - ¿Cuántos dígitos de paridad se necesitan?
   - ¿En qué posiciones?
   - ¿Cuáles son los dígitos asociados a cada uno de los de paridad?

9. Para el código de Hamming para mensajes de 4 bits introducido en el apartado 2.3.3, se desea transmitir la información 1001.
   - Cifre el mensaje que debe enviarse
   - Si consideramos las cinco posibilidades: que el mensaje llegue inalterado a su destino, o que uno de sus cuatro dígitos haya cambiado su valor, descifre cada uno de esos mensajes.

10. Para cifrar un código de Hamming para mensajes de 11 bits, construimos un vector de 15 bits.

    - Dé fórmulas apropiadas para hallar la paridad de los bits insertados.
    - Ídem para la autocorrección que se efectúa en el descifrado.

2.5 Comentarios bibliográficos

El material incluido en este capítulo sobre la representación de los números reales en coma flotante sólo es una aproximación conceptual. En [Gol91] se encontrarán muchos de los detalles técnicos omitidos aquí, tales como el tratamiento dado en la práctica a los errores (absolutos y relativos) debidos al redondeo, el manejo de las excepciones producidas por el desbordamiento y su concreción en los sistemas normalizados por la IEEE.

Aunque el sistema de numeración sexagesimal no puede considerarse posicional (tal como lo usamos para medir el tiempo o los ángulos) por expresarse el peso asociado a las "cifras" de una cantidad explícitamente, y no mediante su posición, este sistema es probablemente el precursor de los sistemas posicionales. Para completar la referencia histórica, debe decirse que, en la antigua Babilonia, también se conocía un sistema de coma flotante, que seguramente es el primero de esta clase [Knu72].

En [For70, Rum83, KM86] puede encontrarse una gran diversidad de ejemplos sobre la aparición de discrepancias intolerables entre los resultados teóricos y los hallados en diversas máquinas.

Los convenios presentados en este capítulo constituyen tan sólo una pequeña parte de los ideados para mantener o proteger la información. Se han escogido algunos de los códigos más ilustrativos y los más difundidos, aunque faltan otros, tales como el llamado *binario reflejado* y los de Gray. Una introducción a los mismos puede consultarse en [Mei73].

Un aspecto de gran interés relacionado con la protección de la información es la criptología, que estudia mecanismos para ocultarla, cifrándola en claves secretas (criptografía), así como para descifrarla (criptoanálisis). En [Dew88, Dew89] puede encontrarse una sencilla introducción a estos temas.

Capítulo 3

Estructura física de un computador

El principal objetivo de este tema es introducir algunos conceptos básicos acerca de la estructura física (hardware) de un computador; pero ¿es de verdad útil preocuparse por la estructura interna de un computador?

Entre las dos posiciones extremas (el simple usuario y el informático profesional) se encuentra una gran cantidad de profesionales que requieren conocer los computadores con un cierto detalle. Al menos, siempre es necesario conocer las características del computador y los requerimientos (físicos) de los programas que deben usarse.

En las siguientes secciones estudiamos el hardware de una computadora, formado por la UCP, la memoria y los periféricos (también llamados dispositivos de entrada y salida, E/S).

El significado del término *hardware* no es fácil de expresar en español con una sola palabra; literalmente se debe entender como "conjunto de útiles duros"; en el contexto que nos ocupa, el hardware de un computador es el conjunto de dispositivos físicos que lo componen, mientras que otra palabra inglesa, *software*, designa los programas que puede ejecutar el computador.

En cierto modo, el hardware es comparable al cerebro o, más generalmente, al cuerpo físico del computador mientras que el software sería lo equivalente a las ideas que pueblan el cerebro. Es conveniente señalar, a pesar de su evidencia, que el hardware y el software son perfectamente inútiles aisladamente: de nada nos sirve un computador si no tenemos ningún programa que ejecutar, y de nada nos sirve tener muchos programas si no disponemos de un computador que los ejecute.

Nuestra visión del sistema formado por el hardware y el software es funcional, y en la última parte del capítulo será patente esta relación de dependencia mutua: explicaremos el funcionamiento del hardware siguiendo la ejecución de algunos programas sencillos, escritos en su propio lenguaje.

3.1 Componentes de un computador

Para introducir los conceptos básicos que estudiaremos dentro de esta sección consideramos un computador como una unidad de producción. Una unidad de producción adquiere materia prima, la elabora y, finalmente, vende la materia elaborada. Esto mismo es lo que hace un computador: toma algunos datos, los procesa y, finalmente, devuelve el resultado obtenido al procesar la información.

Más concretamente, consideremos una panadería ideal. En esta panadería se compra harina, levadura, ... (entrada de datos) que posteriormente se elaboran (procesamiento) para producir pan que, finalmente, se vende (salida de datos). Para comprar y vender se necesitan personas que se relacionen con el exterior; en un computador esta labor se realiza mediante los *periféricos*. Dentro de la panadería podemos encontrar dos zonas bien diferenciadas e indispensables: la primera es la zona

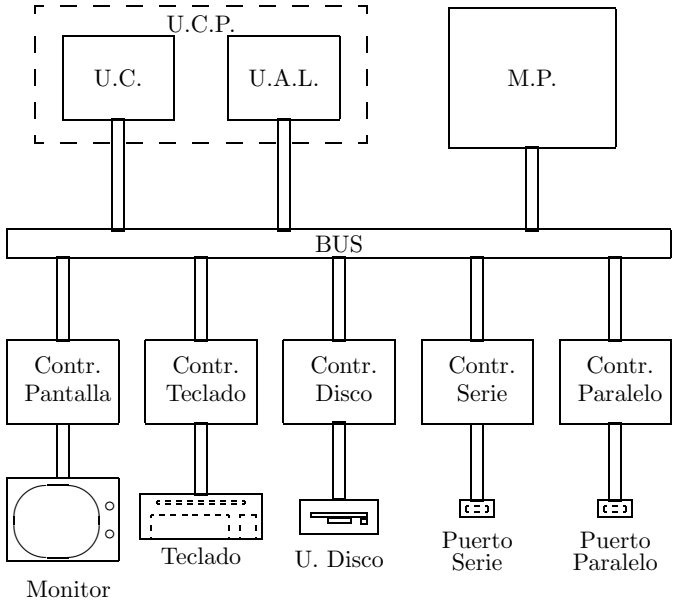


Figura 3.1. Estructura básica de un computador.

de amasado junto con el horno, y la segunda es el almacén.

En el computador el almacén lo representa la *memoria* y la zona de trabajo es la *unidad central de proceso* o UCP que, a su vez, consta de *la unidad de control* o UC (el encargado que controla los procesos de amasado y horneado) y la *unidad aritmética y lógica* o UAL (zona de amasado y horno). Naturalmente, entre las distintas zonas deben existir pasillos de comunicación para poder sincronizar las acciones de cada uno; en un computador esta información se envía y recibe a través de los *buses*.

En las siguientes secciones estudiaremos cada una de las partes que componen la estructura física de un computador. La figura 3.1 presenta un esquema de la misma.

3.1.1 Memoria principal

Mencionábamos en el párrafo anterior que la memoria representa el almacén donde se guarda la información, en esta sección estudiaremos algunos detalles del almacenamiento de información en la memoria y de los tipos de memoria existentes.

En la memoria principal se guarda el conjunto de instrucciones (programa) que está siendo ejecutado, junto con los datos de entrada y de salida de la ejecución. Estudiaremos la memoria de un computador desde un punto de vista físico (distintos medios de almacenamiento) y desde un punto de vista lógico (de tratamiento de la información).

Podemos encontrar similitudes entre la organización física de la memoria y el almacén de la panadería ideal que introducíamos al principio del capítulo: en el almacén encontramos estanterías repletas de bandejas iguales, y cuando un trabajador entra en el almacén, bien trae o bien retira algunas de estas bandejas. La unidad mínima a la que se accede no es una barra de pan (bit) sino una bandeja completa (palabra). La longitud de *palabra de memoria* viene representada por la capacidad de cada bandeja.

No podemos acceder a cada bit de la memoria aisladamente; la mínima cantidad de memoria a la que podemos acceder está formada por una palabra de memoria. Físicamente, la memoria está dividida en celdas (con una capacidad de información de un bit), agrupadas en palabras de memoria.

Funcionamiento de la memoria

Para acceder a cada palabra de memoria debemos poder referirnos a ellas. Esto se hace asignando una dirección numérica binaria a cada palabra a modo de "dirección postal". La dirección de memoria determina una palabra de memoria, que es la que contiene la información.

Supongamos que tenemos un computador que tiene palabras de memoria de 1 byte (8 bits) y dispone de 1 Mb (2^{20} bytes) de memoria principal. Cada dirección de memoria tiene asignado un número en binario

entre 0 y $2^{20} - 1$. Para poder denotar todas las palabras de memoria disponibles necesitaremos al menos 20 dígitos ya que, en general, con d bits es posible direccionar 2^d palabras de memoria. Téngase en cuenta que, en principio, la longitud de palabra de memoria no tiene relación con la longitud de las direcciones de memoria; en el apartado sobre optimización de memorias tendremos la oportunidad de profundizar algo más sobre ello.

En la memoria se realizan operaciones elementales de lectura y escritura, que escriben o leen la información contenida en una sola palabra de memoria. Tanto la UCP como la memoria se sirven de unas cuantas palabras de acceso muy rápido, llamadas *registros*. Para las operaciones de lectura y escritura, los dispositivos de memoria disponen de dos registros: el *de dirección* (RD) y el *de intercambio de memoria* (RIM).

El RD indica la dirección de memoria que se quiere leer o en la que se quiere escribir; puesto que debe tener capacidad para albergar cualquier dirección de memoria, es un registro de d bits siguiendo la notación anterior. Por su parte, el RIM alberga la palabra leída o que se va a escribir en la dirección dada por el RD y, por lo tanto, tiene tantos bits como la longitud de palabra de memoria. La memoria está conectada con la UCP y con los periféricos a través de los *buses* de direcciones, de datos y de control que describiremos más adelante.

El proceso de lectura o escritura se puede dividir en los siguientes pasos:

1. A través del bus de direcciones llega un número de dirección de memoria que se almacena en el RD.

2. Simultáneamente, por el bus de control, llega una señal que indica si la operación que debe realizarse es de lectura o de escritura.

3. Si la operación es de escritura, por el bus de datos llega la palabra que se quiere escribir. Ésta se almacena en el RIM y se escribe donde indique RD. Si la operación es de lectura se lee la información que se encuentra en la dirección almacenada en el RD y se escribe en el RIM.

4. La memoria genera, por el bus de control, una señal de control que indica el fin de la operación.

Clasificación de las memorias

La memoria se encarga de intercambiar información con el procesador según las necesidades de éste. Con la tecnología actual los procesadores alcanzan velocidades de varios millones de cómputos por segundo, lo que obliga a la memoria a tener una velocidad semejante a fin de no menguar la eficiencia del computador. Por otra parte, la capacidad de memoria es otra característica interesante, ya que, en principio,[1] no podríamos ejecutar programas que no pudieran ser cargados completamente en memoria. Las características de velocidad y capacidad están reñidas entre sí, por lo que es necesario alcanzar un compromiso entre ambas dependiendo de la finalidad. Según el nivel de compromiso alcanzado podemos destacar varios niveles jerárquicos de memoria: de más rápida y cara (por lo que suelen tener menor capacidad) a menos rápida y mayor capacidad:

1. Memoria principal
2. Memoria secundaria
3. Memoria auxiliar

Los dos últimos tipos de memoria serán desarrollados en la sección de periféricos, pues pueden ser considerados como periféricos de almacenamiento.

Hemos dicho anteriormente que las memorias son dispositivos de lectura y escritura, y esto es cierto cuando hablamos de la memoria principal de un computador. Sin embargo, existe otro tipo de memorias de sólo lectura que hace las veces de manual de comportamiento de la máquina que la contiene.

[1] En realidad, cada programa tendrá unos requerimientos mínimos de memoria, aunque ello no significa que tenga que caber completo en la memoria.

Las memorias de sólo lectura reciben el nombre genérico de memorias ROM (acrónimo del inglés *Read Only Memory*). Los computadores vienen dotados con una memoria de este tipo donde se almacena la rutina de arranque. Otras aplicaciones de este tipo de memoria las encontramos en las lavadoras automáticas (los programas de lavado se almacenan en una ROM) y en los juguetes electrónicos. En este tipo de memoria la información es almacenada de forma permanente.

Algunas variantes de la memoria ROM son la PROM (ROM programable), EPROM (PROM borrable, *erasable* PROM) y la EEPROM (PROM eléctricamente borrable). Estos tipos de memoria son útiles en la fase de desarrollo de un sistema, en la cual aún no se ha fijado el contenido final de la ROM.

Las memorias de lectura y escritura suelen llamarse memorias RAM (del inglés *Random Access Memory*). Dentro de este tipo de memorias podemos distinguir las RAM estáticas y las RAM dinámicas. Las estáticas se caracterizan por tener un *tiempo de acceso*[2] igual a cada dirección de memoria (cada dirección tiene su propio camino de acceso dentro del *microchip*, generalmente construido con semiconductores). Por otra parte, son memorias volátiles en el sentido de que necesitan la alimentación eléctrica para conservar la información.

En las RAM dinámicas la información necesita ser recordada periódicamente, ya que se va descargando con el tiempo. Esta pérdida de información es debida a que están construidas usando pequeños condensadores. La razón de ser de este tipo de memorias es la economía, pues generalmente son más baratas que las estáticas.

Optimización de memorias

Sabiendo la necesidad de contar con memorias cada vez más rápidas y con más capacidad se han ideado métodos de optimización para la memoria principal. Dependiendo del aspecto por optimizar encontramos las

[2]El tiempo requerido para leer o escribir una palabra de memoria. En una memoria estática puede ser de unos 20 ns, mientras que en una dinámica puede alcanzar los 80 ns.

memorias caché y la *memoria virtual* que, respectivamente, aumentan la rapidez y la capacidad de la memoria principal.

Las *memorias caché* son memorias hasta mil veces más rápidas que las usuales pero, debido a su alto coste, suelen tener una capacidad muy pequeña. La idea que define las memorias caché no puede ser más simple: se trata de guardar en registros los contenidos de las posiciones de memoria de uso más frecuente, de modo que sea mucho más rápido acceder a la información que hay en ellas.

El funcionamiento de la memoria caché ejerce una acción de filtro sobre las direcciones de memoria que solicita la UCP. La secuencia de acciones que se producen en las operaciones de lectura y escritura con memoria caché son las siguientes:

1. La UCP genera una dirección de memoria que se envía a las memorias principal y caché.

2. Si la dirección se encuentra en la caché, será ésta la que devuelva el dato e inhiba la salida de la memoria principal. De lo contrario, será la memoria principal la que dé el dato.

3. Finalmente, se actualizan las direcciones y los datos contenidos en la caché.

Existen distintas estrategias para seleccionar las direcciones que se guardan en la memoria caché de modo que se mantengan en ella las direcciones más usadas (obsérvese que el conjunto de direcciones de memoria más utilizadas variará con la fase del programa que se esté ejecutando). Una buena estrategia de selección puede conllevar una tasa de aciertos (la dirección requerida está en la caché) muy elevada, de donde la velocidad aparente de la memoria se asemejará mucho a la velocidad de la memoria caché.

La *memoria virtual* se desarrolla con el propósito de poder hacer uso de más memoria de la que físicamente se dispone. Si un programa es demasiado grande para la memoria disponible se solía dividir en módulos que cupieran en memoria mediante la técnica del *solapamiento* (*overlay*). Esta técnica tiene el serio inconveniente de que los programas no

son transportables, puesto que, en general, no funcionarían en un computador con menos memoria.

Para solventar estos problemas se desarrolló la *memoria virtual* como un método automático para realizar el solapamiento. La idea consiste en usar la memoria secundaria, generalmente un disco duro, como memoria principal. Un programador que dispone de memoria virtual tiene la impresión de estar trabajando con un mapa de direcciones de memoria (direcciones lógicas) mucho mayor del que físicamente dispone (direcciones físicas).

Existen otras técnicas para mejorar la utilización de la memoria como son la *paginación* y la *segmentación*. Estas técnicas utilizan programas de gestión de memoria que forman parte del sistema operativo; en el capítulo 4 se presentarán más detalles sobre ellas.

3.1.2 Unidad central de proceso

La UCP representa el cerebro de la computadora y allí es donde se procesa la información recibida, por lo que casi siempre nos referiremos a ella como el *procesador*. La UCP está formada por la *unidad de control* (UC), que clasifica y organiza las instrucciones recibidas (encargado), y la *unidad aritmética y lógica* (UAL), que las ejecuta (donde se "amasa y cuece" la información).

Físicamente el procesador es un *microchip* y consta de unos circuitos electrónicos que permiten realizar operaciones elementales con la información. El procesador se conecta con el resto de los componentes de un computador mediante unas patillas metálicas, cada una de las cuales transporta información binaria, a través de los buses de comunicación que estudiaremos más adelante.

El cometido de la UC consiste en recibir la instrucción que se va a ejecutar, determinar su tipo (cálculo aritmético, lógico, ...), determinar si esa instrucción necesita argumentos almacenados en la memoria, leer (en su caso) las direcciones de memoria que contienen los argumentos de la instrucción y dar la orden correspondiente a la UAL. Por su parte, la labor de la UAL es la de ejecutar las instrucciones aritméticas

y lógicas, una vez que la UC ha determinado su tipo y ha leído sus argumentos (si los hubiera). Las instrucciones que llegan a la UAL son muy sencillas, y se reducen a un cálculo aritmético elemental (según el tipo de procesador, "elemental" significará bien suma-resta o bien suma-resta-multiplicación-división), un cálculo lógico (*and, or*, ...), o una instrucción de salto o bifurcación.

Unidad de control

La UC se encarga de clasificar las instrucciones que recibe, controlar su ejecución y leer las zonas de la memoria que almacenan los argumentos de estas instrucciones. La UC está dotada de unos cuantos registros internos de memoria que usa para almacenar datos elementales durante la ejecución de una instrucción elemental. Esta memoria dispone de un cierto número de registros con un cometido particular, entre los que destacan el *registro de instrucción* y el *contador de programa*.

El registro de instrucción almacena aquélla que está siendo ejecutada y, por su parte, el contador de programa almacena la dirección de la siguiente instrucción que debe ser ejecutada. Existen, además, otros registros que almacenan los resultados parciales de la ejecución de una instrucción.

El funcionamiento de la UC está regido por los impulsos de un reloj que sincroniza la realización de las distintas operaciones y determina la velocidad del procesador. Su frecuencia se mide en *MHz* (*megaherzios*, millones de ciclos por segundo).

El trabajo desempeñado por la unidad de control al ejecutar una instrucción puede descomponerse en pequeños pasos como los descritos a continuación:

1. Leer el contador de programa.

2. Almacenar en el registro de instrucción el contenido de la dirección de memoria que aparece en el contador de programa.

3. Averiguar si la instrucción necesita argumentos y, en su caso, determinar sus direcciones de memoria.

4. Leer los argumentos y almacenarlos en los registros internos.

5. Ordenar a la UAL que ejecute el cómputo necesario.

6. Almacenar el resultado de la ejecución.

7. Actualizar el contador de programa con la siguiente instrucción por ejecutar.

La UC dispone de un dispositivo denominado *secuenciador* que efectúa esta descomposición en pasos elementales.

Unidad aritmética y lógica

La unidad aritmética y lógica es el horno donde se cuece la información; su tarea consiste en recibir instrucciones junto con sus argumentos y ejecutarlas, dando a cambio el resultado de su operación.

Esta unidad consta de un(os) operador(es) que ejecuta(n) físicamente las instrucciones recibidas, una serie de registros para almacenar información mientras se ejecuta una instrucción (entre estos registros destaca el registro *acumulador*, al que se hará referencia de nuevo cuando estudiemos el direccionamiento de las instrucciones) y algunos señalizadores de estado que indican resultados interesantes obtenidos al realizar un cómputo (resultado cero, *overflow* o desbordamiento, ...)

Los operadores son dispositivos físicos (circuitos electrónicos) que pueden realizar operaciones elementales sobre datos binarios. Las operaciones que son capaces de hacer estos dispositivos pueden ser de desplazamiento, lógicas o aritméticas.

Las operaciones de desplazamiento consisten en desplazar los bits de una palabra varios lugares hacia la izquierda o hacia la derecha. Dependiendo de la acción del desplazamiento sobre los extremos de la palabra podemos distinguir varios tipos de desplazamiento:

1. *Desplz. lógico*: si el extremo de la palabra que queda vacío tras el desplazamiento se completa con ceros.

2. *Desplz. aritmético*: es similar al anterior, pero se mantiene el bit de signo. Se utiliza para representar multiplicaciones y divisiones de una potencia de 2.

3. *Desplz. circular*: los bits que quedan fuera tras el desplazamiento se emplean en llenar los huecos libres del otro extremo de la palabra.

4. *Desplz. concatenado*: se desplaza conjuntamente el contenido de dos o más registros.

Las operaciones lógicas tales como NOT, AND y OR se realizan bit a bit. La primera de estas operaciones sólo depende de un argumento, mientras que las restantes necesitan dos argumentos.

Las operaciones aritméticas más importantes que se realizan en la UAL son las de *suma, resta, multiplicación* y *división*, la de *cambio de signo* y la de *extensión de signo*. Esta última operación se hace necesaria cuando se transmite información a un elemento con mayor longitud de palabra pues es necesario completar los bits restantes sin alterar la información. En general las operaciones de multiplicar y dividir se hacen usando sumas y restas mediante un algoritmo apropiado; sólo computadores muy potentes (y caros) disponen de operadores particulares que las realicen directamente.

Se puede mejorar la capacidad de cálculo numérico de algunos procesadores añadiendo un *coprocesador* matemático. Los dispositivos de este tipo complementan la UAL del procesador por otra más potente; con mayores y más numerosos registros operativos, con una representación interna de los datos de mayor precisión y con instrucciones numéricas más complejas (funciones exponenciales, logarítmicas y trigonométricas).

Para ello, comparten el flujo de instrucciones y datos del procesador y cuando detectan alguna instrucción numérica toman el control del programa, ejecutan la instrucción, calculan el resultado y devuelven el control al procesador.

3.1.3 Periféricos

Damos en esta sección una visión general de los periféricos. Podemos considerar periféricos de entrada, de salida y de almacenamiento;

asimismo podemos distinguir entre periféricos *locales* y periféricos *remotos*, según su conexión al computador. Un periférico local, como el ratón, se encuentra cerca de la UCP conectado mediante cables que hacen las veces de prolongador de los buses del computador. Para un periférico remoto, como una impresora láser del centro de cálculo, la conexión se realiza a través de una red de comunicaciones.

A continuación se enumeran algunos de los periféricos más importantes:

De entrada de datos

1. *Teclado.* Es similar al teclado de una máquina de escribir y cuenta además con algunas teclas de control.

2. *Ratón.* Es un dispositivo que al ser desplazado sobre una superficie permite mover el cursor por la pantalla. Existen ratones de sistema mecánico y de sistema óptico.

3. *Sensores.* Este tipo de periféricos incluye a las pantallas táctiles, capaces de seleccionar distintas opciones reconociendo el tacto sobre distintas zonas de la pantalla. También podemos encontrar otro tipo de sensores como

   (a) *Lápiz óptico.* Cuando se posa en la pantalla reconoce la posición que ocupa mediante una medición de la luminosidad que recibe.

   (b) *Tableta gráfica.* Similar a una pizarra provista de un lápiz. Los trazos sobre la tableta aparecen en la pantalla del computador.

4. *Escáner.* Permite digitalizar imágenes planas (fotografías o texto) y archivarlas.

De salida de datos

1. *Pantalla o monitor.* Es el principal instrumento de comunicación entre el computador y el usuario. Su constitución física es similar

a la del tubo de imagen de un televisor. Es usual llamar *consola* al conjunto formado por un teclado y un monitor.

2. *Plotter.* Permite realizar gráficos de alta precisión como mapas o diseños técnicos.

3. *Impresora.* Su misión es proporcionar copias impresas en papel de la información guardada en el computador. Hay diversos tipos de impresora, entre los que destacan las impresoras de margarita (ya en desuso), de matriz de puntos, de chorro de tinta y las impresoras láser.

De entrada y salida de datos

1. *Módem.* Es un dispositivo que permite la comunicación de un computador con otro a través de la línea telefónica (red conmutada) o a través de líneas destinadas exclusivamente a este fin (líneas punto a punto). Para ello convierte los datos binarios en señales moduladas de baja frecuencia. Existen diversos protocolos que determinan la forma de iniciar, efectuar y finalizar la transmisión, así como su velocidad y corrección de errores. Su nombre procede de su doble función: MOdulador, DEModulador. Su velocidad se mide en *baudios*, que equivalen aproximadamente a bits por segundo, al incluir las necesarias señales de control.

2. *Red.* Las redes permiten la interconexión de varios computadores entre sí, la utilización conjunta de distintos dispositivos externos tales como un disco duro, una impresora, etc., y el uso compartido de programas y ficheros de datos. Cada computador conectado a la red contempla los distintos dispositivos disponibles como si fueran propios. Por lo general uno de los computadores se dedica en exclusiva a la gestión de la red, denominándose *servidor de red.* Las redes pueden ser *locales*, cuando se ubican en la misma habitación o edificio, o *remotas.* En general, tanto el sistema operativo como los programas de aplicación son específicos para el funcionamiento en red.

Periféricos de almacenamiento

Aunque se trata en realidad de periféricos de entrada y salida, suelen estudiarse aparte. Los periféricos de almacenamiento son también conocidos como memorias secundarias y memorias auxiliares. La mayoría de estos dispositivos almacenan la información de forma magnética. El primero de todos los dispositivos de almacenamiento magnético fue la unidad (lectora y grabadora) de *cinta magnética*, y posteriormente se desarrollaron las unidades de *discos fijos* (también llamados *discos duros*) y las unidades de *discos flexibles*.

En una cinta magnética el acceso a la información es *secuencial* (tenemos que hacer correr la cinta hasta que aparezca la información que buscamos); esto hace que sea un medio muy lento. Generalmente las cintas magnéticas, debido a su gran capacidad, se utilizan para hacer periódicamente copias de seguridad (*backup*) de la información almacenada en los discos duros del computador. Recientemente se han desarrollado los *streamers*, que son dispositivos cuya única finalidad es hacer copias de seguridad de grandes volúmenes de información, generalmente contenida en un disco duro.

Los discos magnéticos reciben este nombre por su forma y porque su superficie es magnética (ciertamente no es un nombre muy original, aunque sí autoexplicativo) y son dispositivos de *acceso directo*, esto es, no tenemos que recorrer toda la información que hay delante de la que necesitamos.

Los discos magnéticos necesitan organizarse lógicamente para poder albergar información de un modo ordenado; dar formato a un disco magnético es dotarlo de la organización lógica necesaria para cada modelo de computador (no es lo mismo el formato del DOS, que usan los compatibles con IBM, que el formato usado por los computadores *Macintosh*).

La información se almacena siguiendo círculos concéntricos llamados *pistas* que a su vez se dividen en *sectores* que contienen un cierto número de palabras (celdas). Para indicar una dirección se especifica la pista y el sector donde comienza la información, por lo cual una transferencia

de datos a un disco siempre empieza en la primera palabra de un sector. Las operaciones necesarias para dar formato a un disco magnético son bastante complejas y, por lo tanto, se suele dejar al sistema operativo la gestión de todas las pequeñas tareas que hay que llevar a cabo, en el capítulo 4 veremos que el sistema operativo es el software encargado de facilitar este tipo de tareas a los usuarios del computador.

Entre los distintos tipos de disco destacan los discos duros (*hard disk*), que disponen de una gran capacidad de almacenamiento (de 20 a 800 Mb). Suelen ser fijos (no se pueden extraer del computador) y suelen contener el sistema operativo y los programas de uso más común.

Los discos flexibles, *diskettes* o disquettes (en inglés, *floppies*), son extraíbles y por eso pueden ser usados para transferir información de un computador a otro (que pueda leer discos flexibles). Estos discos tienen una capacidad mucho menor que un disco duro (entre 360 Kb y 2'88 Mb) y el tiempo de acceso a la información almacenada es bastante grande. Actualmente coexisten discos flexibles de dos tamaños distintos, de $5\frac{1}{4}''$ y de $3\frac{1}{2}''$ (pulgadas), de similares prestaciones. Los computadores compatibles con IBM pueden utilizar discos flexibles de $5\frac{1}{4}''$ y de $3\frac{1}{2}''$, mientras que los computadores de la familia *Macintosh* usan exclusivamente los de $3\frac{1}{2}''$. Este hecho hará que, probablemente, los discos de $5\frac{1}{4}''$ acaben desapareciendo del mercado.

Últimamente se han desarrollado las unidades de discos duros extraíbles, que tienen las ventajas de los discos duros en cuanto a capacidad y a velocidad de acceso y además son intercambiables.

3.1.4 Buses de comunicación

Los componentes principales de un computador son la UCP, la memoria y los periféricos. Estas componentes intercambian información constantemente y, obviamente, la comunicación debe establecerse a través de un medio físico que conecte la UCP con los dispositivos E/S y con el exterior. La comunicación entre los distintos componentes se realiza a través de líneas que transportan información binaria. Este transporte puede llevarse a cabo de dos modos:

- En la comunicación en *serie*, la información se transmite un bit tras otro. El ratón es un ejemplo típico de periférico con comunicación en serie.

- En la comunicación en *paralelo*, se transmite la información a través de varias líneas simultáneamente, de modo comparable a una autovía de varios carriles (líneas) por las que los vehículos (información binaria) fluyen simultáneamente.

  En la comunicación en paralelo con los periféricos hay ocho o nueve líneas, y se transmite de byte en byte. Un periférico típicamente comunicado en paralelo es la impresora.

Las líneas de comunicación se agrupan según el tipo de información que transporten, y cada uno de estos conjuntos de líneas recibe el nombre de *bus*. Un bus transmite la información en paralelo.

Atendiendo al tipo de información que transmiten, los buses de comunicación pueden ser de tres clases:

- *Bus de direcciones,* a través del cual la UC determina la dirección de memoria o dispositivo de E/S con el que se intercambia información.

- *Bus de datos,* por el que viajan los datos para ser almacenados en la memoria o para ser usada en algún cómputo.

- *Bus de control,* que, como su nombre indica, transporta información de control para la sincronización de todo el trabajo.

En general los buses de datos tienen un número de líneas igual a la longitud de la palabra de máquina, aunque a veces sólo tienen la mitad, lo que incide negativamente en la velocidad del computador pero suele abaratar el precio. Si el número de celdas de memoria accesibles es 2^d, los buses de direcciones suelen tener d líneas. Por último, los buses de control tienen un número de líneas variable dependiendo de las distintas marcas y modelos de procesador.

Por el bus de datos la UC recibe (el código binario de) una instrucción, la interpreta y prepara su ejecución. Dependiendo del tipo de

instrucción, la UC puede generar algunos códigos de control que serán enviados a través del bus de control; en su caso, averiguará a través del bus de direcciones en qué posición (de la memoria principal) encuentran los argumentos de la instrucción, esta información viajará hasta la UC por el bus de datos y, finalmente, el resultado del cómputo será transportado de nuevo a través del bus de datos hacia la memoria o hacia un periférico.

Existen dispositivos electrónicos para el control de los buses que, en ciertos casos, liberan a la UCP de este trabajo: son los *controladores*.

- Los controladores *del sistema* permiten el traslado del contenido de bloques de memoria, a gran velocidad, a través del bus, con independencia de la UCP. Se permite así el acceso directo a la memoria de los periféricos que puedan precisarlo (pantalla y discos duros).

- Los controladores *de dispositivo* actúan como intermediarios entre los periféricos y los buses permitiendo la comunicación entre la UCP y sus periféricos, dado que estos últimos no se conectan directamente a los buses. Pueden ser especializados, como los que controlan la pantalla o las unidades de disco, o de propósito general, bien en *serie* o en *paralelo*.

La comunicación entre la UCP y los controladores de dispositivo se puede hacer de dos formas distintas: utilizando la propia memoria principal o través de una memoria independiente para E/S. En el primer caso cada controlador de dispositivo tiene asignada una dirección de memoria. Cuando la UCP quiere leer o escribir en el dispositivo, lo hace en la dirección que dicho dispositivo tiene asignada, utilizando las mismas instrucciones de escritura o lectura de memoria. En el segundo caso, el lenguaje máquina debe disponer de instrucciones especializadas para el acceso a este área de E/S.

Los controladores de propósito general cumplen un determinado *protocolo* estándar de comunicaciones y disponen de los necesarios conectores en el exterior del computador. Cualquier periférico que se atenga a dicho protocolo puede ser conectado a ellos, facilitando su utilización.

Entre los protocolos más extendidos cabe citar el *RS*-232 para los puertos en serie y el *Centronics* para los puertos paralelos.

3.2 Instrucciones en lenguaje de máquina

La información se representa dentro de un computador mediante ceros y unos. Cada procesador es capaz de distinguir si recibe una instrucción o un dato de una forma que depende de su marca y su modelo.

El juego de instrucciones de un procesador recibe el nombre de *lenguaje de máquina* o *código máquina*. Una UCP sólo puede entender instrucciones expresadas en su lenguaje de máquina, y cada instrucción especifica una acción particular sobre algunos operandos. Una instrucción es una lista de ceros y unos: una parte de la lista es el *código de la operación* que ha de realizarse, el resto determina dónde se encuentran los argumentos de la instrucción (si los hubiera). Todo lo relacionado con la especificación de dónde están los argumentos de la instrucción recibe el nombre de *direccionamiento*. A las instrucciones de máquina se les asignan nombres nemotécnicos, más fáciles de recordar que listas de ceros y unos.

Estas instrucciones son muy elementales, por lo tanto es necesario realizar un gran esfuerzo de traducción entre el lenguaje natural y el código máquina. En el capítulo 5 estudiaremos la jerarquía de niveles que permiten a un programador de aplicaciones escribir sus programas en lenguajes a medio camino entre el lenguaje natural y la codificación en binario.

Hay dos tendencias básicas en el diseño de juegos de instrucciones: pocas instrucciones (algunas decenas) simples pero de ejecución muy rápida o muchas instrucciones (unas 200) complejas, de ejecución algo más lenta. La primera se conoce como RISC (del inglés, *Reduced Instruction Set Computer*: computador con juego de instrucciones reducido) y la segunda como CISC (*Complex Instruction Set Computer*: computador con juego de instrucciones complejo).

3.2.1 Formato de las instrucciones

La elección del formato de las instrucciones de un procesador depende en buena parte de las especificaciones fijadas por el equipo de diseño del procesador y de otras consideraciones que veremos a continuación.

Entre las operaciones que se ejecutan en un procesador, algunas no tienen operandos, otras tienen uno, dos o a lo sumo tres. En muchos de los casos los operandos vienen expresados por sus direcciones de memoria, por lo que se suele hablar de instrucciones de una, dos o tres direcciones.

Pueden existir instrucciones sin operando, bien porque el operando no aparezca explícitamente, siendo uno de los registros del procesador, o porque la propia instrucción no lo necesite, como cuando se repite un proceso o se regresa de una llamada a un subprograma.

En el otro extremo se sitúan las operaciones binarias, que precisan tres direcciones para llevarse a cabo: las de sus dos argumentos y la de dónde ha de colocarse el resultado obtenido.

Lo ideal en un juego de instrucciones es que todas tengan el mismo formato. Adoptar, por ejemplo, un juego de instrucciones de tres direcciones es sencillo: basta con ignorar los argumentos añadidos para igualar el formato.

También es posible elegir un juego de instrucciones con menos de tres operandos. Las operaciones de tres argumentos se traducen entonces como sigue (considerando una operación aritmética cualquiera):

Si el juego de instrucciones es de dos direcciones, éstas representan a ambos argumentos, y el resultado de la ejecución se almacena en la primera o segunda dirección suministrada.

En las instrucciones de una dirección sólo se especifica la de uno de los argumentos. En el caso de que se trate de una instrucción binaria, se hace necesario usar el registro *acumulador* que se encuentra en la UAL. Entonces, se toma como primer argumento el contenido del acumulador, como segundo argumento el contenido de la dirección de memoria dado por la instrucción, y el resultado de la instrucción se almacena en el registro acumulador (ver sección 3.3.2).

3.2.2 Tipos de Instrucciones

Las instrucciones del juego de un procesador pueden ser divididas según su cometido en los siguientes grupos:

1. *De movimiento de datos.* Este tipo de instrucciones transfieren datos entre la memoria principal y los registros. Combinando las distintas procedencias con los distintos destinos posibles obtenemos bastantes tipos de instrucciones de movimiento de datos. Puede tratarse de datos aislados, de bloques de datos o de cadenas de caracteres.

2. *Operaciones binarias.* Estas instrucciones, aritméticas y lógicas, realizan una operación con dos argumentos como, por ejemplo, las operaciones aritméticas elementales y algunas operaciones lógicas binarias tales como AND, OR y XOR.

3. *Operaciones monarias.* Entre estas instrucciones podemos encontrar las que desplazan o rotan los bits de una palabra. Algunas operaciones binarias ocurren tan a menudo con un mismo argumento que, a veces, son incluidas como instrucciones de una sola dirección. Por ejemplo, tenemos la instrucción de borrar el contenido de una palabra de memoria, que es un caso particular de "mover una palabra formada por ceros a la dirección suministrada".

4. *Instrucciones de salto.* Sirven para alterar el orden de ejecución de las instrucciones. Dentro de este grupo encontramos las instrucciones de salto condicional y las de salto incondicional:

   La ejecución de una instrucción de salto incondicional obliga al computador a "saltarse" el orden secuencial para ejecutar la instrucción contenida en la dirección determinada por el argumento de la instrucción de salto, y el orden de ejecución sigue a partir de la instrucción sobre la que se saltó.

   Una instrucción de salto condicional necesita de instrucciones de comparación ya que es necesario realizar una o varias comparaciones para comprobar la condición.

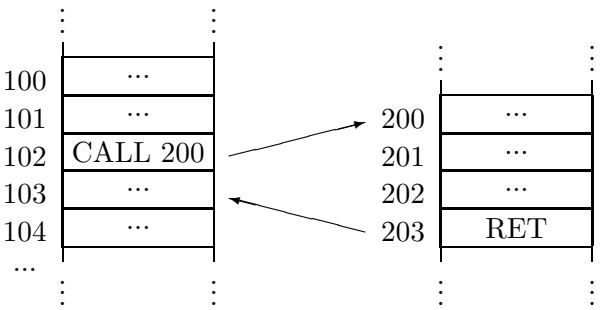


Figura 3.2. Funcionamiento de una llamada a subprograma.

5. *Llamada a un subprograma.* Un subprograma no es más que un grupo de instrucciones que realiza operaciones útiles y necesarias en distintos puntos de un programa. Si, al ejecutar un programa, se encuentra una instrucción de llamada a subprograma, se ejecutan todas las instrucciones del subprograma y posteriormente se pasa a la siguiente instrucción del programa (ver la figura 3.2).

6. *Entrada y salida de datos.* Es el tipo de instrucción que más cambia de un procesador a otro. Sirven para gestionar el intercambio de información entre el computador y el exterior.

3.3 Un ejemplo de recapitulación

En este apartado vamos a desarrollar un ejemplo explicativo de un conjunto de instrucciones de máquina de una dirección con acumulador.

Consideraremos un tamaño de palabra de datos de 16 bits, las instrucciones se codifican con 4 bits, lo que permite un total de 16 instrucciones diferentes. Los restantes 12 bits se utilizarán para codificar las direcciones, lo que permite direccionar $2^{12} = 4096$ posiciones de memoria. El bus de datos será de 16 bits y el de direcciones de 12 bits.

Para simplificar supondremos que el tamaño de palabra de memoria es de 16 bits, es decir que en cada dirección de memoria se almacenan 16

bits, y que en cada operación de lectura o escritura de memoria se trabaja con 16 bits. (En muchos sistemas el tamaño de palabra de memoria es de 8 bits. Por ello, la formación de un dato de 16 bits requiere la lectura de dos posiciones consecutivas de memoria empezando, por ejemplo, por la posición par.)

3.3.1 UCP con acumulador

Como hemos visto (apartado 3.1.1), los registros son memorias de gran velocidad utilizadas por la UCP y la memoria principal para realizar sus operaciones. Su tamaño depende del contenido que vayan a almacenar. En nuestro caso, el registro de dirección de memoria tendría 12 bits y el de intercambio de memoria 16 bits.

El contador de programa tiene que almacenar una dirección de memoria, luego tendrá 12 bits, mientras que el registro de instrucción, que tiene que almacenar la instrucción completa, tendrá 16 bits. Además existirá un registro acumulador, que desempeña en este caso un papel fundamental en la ejecución de instrucciones.

Dado que muchas instrucciones operan sobre dos operandos y producen un resultado, como primer operando se toma el contenido actual del registro acumulador y como segundo operando se toma el contenido de la dirección que aparece en la instrucción y el resultado se almacena nuevamente en el acumulador. Por lo tanto, en las instrucciones de dos operandos el registro acumulador hace el doble papel de operando y de destinatario del resultado.

Supongamos que queremos sumar el contenido de la posición de memoria 100 con el contenido de la 150 y almacenar el resultado en la 200. En primer lugar cargamos el contenido de la posición 100 en el acumulador, a continuación llamamos a la operación suma con la dirección 150; como veremos, esta instrucción suma al acumulador el contenido de la dirección 150. Por último, el contenido del acumulador se almacena en la posición 200 de la memoria.

El proceso sería el siguiente:

```
CARGAR      100
SUMAR       150
ALMACENAR   200
```

Si utilizásemos instrucciones de tres direcciones, simplemente se escribiría:

```
SUMAR 100 150 200
```

El lenguaje máquina se simplifica al tener una única dirección, pero las operaciones se complican al tener que cargar y almacenar los operandos del acumulador.

En las operaciones de cambio de signo o de complementación, solamente tenemos un operando. La operación se realiza sobre el contenido del acumulador, donde se almacena también el resultado. La dirección pasada como argumento no se tiene en cuenta.

En las operaciones de E/S, el contenido de una posición de memoria se recibe o envía a un dispositivo externo. En este caso, no se utiliza el registro acumulador.

3.3.2 Un juego de instrucciones de máquina de una dirección

En la tabla 3.1 se relacionan las instrucciones de un lenguaje de máquina de una dirección simplificado. La notación $M[d]$ representa el contenido de la dirección d de la memoria. La instrucción `CAR`gar d deposita el contenido de la posición de memoria d en el acumulador.

La instrucción `ALM`acenar d realiza el proceso contrario: guarda el contenido del acumulador en la dirección de memoria d.

Las instrucciones `IN` d y `OUT` d se utilizan para E/S. La primera lee un dato de una posición de memoria destinada a un dispositivo externo y lo almacena en la dirección de memoria d; la segunda toma el contenido de la posición de memoria d y lo envía a la dirección de memoria destinada al dispositivo externo.[3]

[3]Por ser este un modelo muy simplificado, no se tiene en cuenta la comunicación directa con el dispositivo.

Instrucción	Efecto	Código de máquina
`CAR` d	Ac $\leftarrow M[d]$	0000
`ALM` d	$M[d] \leftarrow$ Ac	0001
`IN` d	read($M[d]$)	0010
`OUT` d	write($M[d]$)	0011
`SUM` d	Ac $\leftarrow$ Ac + $M[d]$	0100
`RES` d	Ac $\leftarrow$ Ac – $M[d]$	0101
`OP` –	Ac $\leftarrow$ –Ac	0110
`MUL` d	Ac $\leftarrow$ Ac * $M[d]$	0111
`DIV` d	Ac $\leftarrow$ Ac div $M[d]$	1000
`NOT` –	Ac $\leftarrow$ not Ac	1001
`AND` d	Ac $\leftarrow$ Ac and $M[d]$	1010
`OR` d	Ac $\leftarrow$ Ac or $M[d]$	1011
`COND` d	If Ac > 0 then goto d	1100
`GOTO` d	goto d	1101
`END`	Fin de programa	1110

Tabla 3.1. Ejemplo de juego de instrucciones de una dirección

Las instrucciones `SUM`ar *d* y `RES`tar *d*, toman el contenido de la dirección de memoria *d* y lo suman y restan respectivamente con el contenido del acumulador, almacenando el resultado en dicho registro.

El mismo proceso pero aplicando la multiplicación o la división es realizado por `MUL`tiplicar *d* y `DIV`idir *d*.

Las instrucciones `OP`uesto – y `NOT` – producen el cambio de signo (complemento a dos) o la complementación del contenido del acumulador. La dirección a la que se aplican no se utiliza.

Las instrucciones lógicas `AND` *d* y `OR` *d* toman el contenido de la dirección de memoria *d* y del acumulador, efectúan la operación *and* u *or* entre ambos operandos y almacenan el resultado en el acumulador.

La instrucción `GOTO` *d* da un salto incondicional y actúa con independencia del valor del acumulador.

La instrucción `COND`ición *d*, toma el valor del acumulador, comprueba si es mayor que cero, y en caso afirmativo carga en el registro contador de programa la dirección *d*. En consecuencia, la siguiente instrucción que ejecutará el procesador será la contenida en la dirección *d*, produciéndose una ruptura de la secuencia de ejecución. Si no se cumple la condición prosigue la ejecución secuencial.

Por último, la instrucción `END` señala el final del programa.

3.3.3 Ejecución de una instrucción. Detalle

La ejecución de cada instrucción de máquina conlleva una serie de pasos elementales y transferencias de datos de unos órganos a otros de la UCP.

Estos pasos elementales se denominan *microinstrucciones* y son ejecutados por el secuenciador a partir del código de la instrucción de máquina.

La ejecución de una instrucción comienza cuando se actualiza el registro contador de programa. La UC envía esta dirección a los circuitos de selección de memoria a través del bus de direcciones y la señal de

lectura a través del bus de control. La dirección queda almacenada en el registro de dirección de memoria y al recibir la señal de lectura, se lee la instrucción. La instrucción se envía a través del bus de datos y se recibe en el registro de instrucción de la UC. De esta forma, la UC tiene disponible la instrucción para ser decodificada y ejecutada.

Supongamos que la instrucción para ejecutar sea:

$$\texttt{SUM}\ 1000_{(10}$$

equivalente en binario a:

0100	001111101000

La UC separa el código de operación (0100) de la dirección del operando (001111101000) y, a partir del código, genera las señales de control para enviar la dirección, a través del bus de direcciones, al registro de dirección de la MP, efectuar la operación de lectura y, finalmente, enviar el contenido de la dirección 1000 a un registro operativo de la UAL, a través del bus de datos.

A continuación se envía la señal de control correspondiente a la suma a la UAL, quien suma al contenido del acumulador el valor del registro operativo, quedando el resultado almacenado en el acumulador.

3.3.4 Traducción y ejecución de un programa sencillo

Vamos a desarrollar un fragmento de programa para elevar un número n a una cierta potencia $a \in \mathbb{N}$. El programa devolverá el valor 1 si a es cero y n^a si es $a > 0$.

Tenemos que reservar un espacio de almacenamiento para los datos, resultados y constantes. Llamaremos n a la base, a al exponente y r al resultado. Usaremos una posición adicional de memoria para almacenar la constante 1 que usaremos para decrementar la potencia.

La descripción del proceso sería la siguiente:

Datos	Dirección	Contenido
n	$d1$	n
a	$d2$	a
r	$d3$	r
1	$d4$	1
	$d5$	multiplicar r por n
		decrementar a
		si $a > 0$ ir a la dirección $d5$
		ir a la dirección $d7$
(entrada)	$d6$	asignar a r el valor 1
		si $a > 0$ ir a la dirección $d5$
	$d7$	continuar programa

La entrada del proceso es la dirección $d6$.

Para traducirlo a lenguaje máquina tenemos que fijar una dirección inicial, por ejemplo la 100, y convertir cada instrucción en sus equivalentes, utilizando el acumulador:

Dirección	Contenido
100:	n
101:	a
102:	r
103:	1
104:	`CAR` 102
105:	`MUL` 100
106:	`ALM` 102
107:	`CAR` 101
108:	`RES` 103
109:	`ALM` 101
110:	`COND` 104
111:	`GOTO` 116
112:	`CAR` 103
113:	`ALM` 102
114:	`CAR` 101

(*pasa a la página siguiente*)

(*viene de la página anterior)*

Dirección	Contenido
115:	`COND` 104
116:	Continuar programa.

El programa comienza a ejecutarse a partir de la dirección 112, realizando la comparación $a > 0$. Veamos un ejemplo de ejecución del programa para los valores $n = 5$ y $a = 0$:

Dirección	Instrucción	Acum.	a (101)	r (102)
			0	?
112:	`CAR` 103	1		
113:	`ALM` 102			1
114:	`CAR` 101	0		
115:	`COND` 104			
116:	Continuar programa			

Veamos otro ejemplo con $n = 5$ y $a = 3$:

Dirección	Instrucción	Acum.	a (101)	r (102)
			3	?
112:	`CAR` 103	1		
113:	`ALM` 102			1
114:	`CAR` 101	3		
115:	`COND` 104			
104:	`CAR` 102	1		
105:	`MUL` 100	5		
106:	`ALM` 102			5
107:	`CAR` 101	3		
108:	`RES` 103	2		
109:	`ALM` 101		2	
110:	`COND` 104			
104:	`CAR` 102	5		
105:	`MUL` 100	25		

(*pasa a la página siguiente)*

(*viene de la página anterior)*

Dirección	Instrucción	Acum.	a (101)	r (102)
106:	`ALM` 102			25
107:	`CAR` 101	2		
108:	`RES` 103	1		
109:	`ALM` 101		1	
110:	`COND` 104			
104:	`CAR` 102	25		
105:	`MUL` 100	125		
106:	`ALM` 102			125
107:	`CAR` 101	1		
108:	`RES` 103	0		
109:	`ALM` 101		0	
110:	`COND` 104			
111:	`GOTO` 116			
116:	Continuar programa.			

El resultado, 125, queda almacenado en la dirección 102.

3.4 Observaciones complementarias

El conjunto de instrucciones de máquina presentado como ejemplo está muy simplificado. Los lenguajes de máquina reales pueden tener hasta cientos de instrucciones, la mayoría de ellas con distintas modalidades de direccionamiento.

Los procesadores disponen también de numerosos registros operativos auxiliares, que se utilizan en la ejecución de las instrucciones y en la formación de las direcciones sobre las que se opera.

Para tener una panorámica más amplia de los lenguajes de máquina conviene conocer los distintos modos de direccionamiento y las instrucciones para la creación de subprogramas.

También son interesantes las instrucciones del tratamiento de las operaciones de E/S y de las situaciones de error o desbordamiento.

3.4.1 Tipos de direccionamiento

Cada procesador dispone de unas reglas precisas y determinadas para la definición de las direcciones o registros donde se encuentran los datos. Estas reglas constituyen los modos de direccionamiento del procesador y pueden llegar a ser bastante complejas. Un procesador puede tener decenas de modos de direccionamiento diferentes, en los que participan registros especializados.

Se llama *dirección absoluta* al valor numérico que cada posición de memoria tiene asignado y por el cual se accede a ella. En general, la dirección contenida en las instrucciones de lenguaje máquina no es la dirección absoluta, sino que ésta se forma desplazando una cierta dirección llamada *dirección de base*, contenida en un registro especializado.

Se llama *dirección efectiva* a la dirección desde la cual, una vez realizadas las necesarias operaciones sobre la dirección contenida en la instrucción, se toman los datos.

Un registro puede contener un dato para operar con él, pero también puede contener una dirección donde se encuentre el dato. (En los ejemplos siguientes supondremos un lenguaje máquina de dos direcciones.)

- *Direccionamiento inmediato*

  El direccionamiento inmediato consiste en incorporar el dato constante a la instrucción. En realidad no es un direccionamiento propiamente dicho, puesto que se dispone del dato:

  $$\texttt{CAR}\ \ AX,\ \ 500$$

  Esta instrucción carga el valor constante 500 en el registro acumulador.

- *Direccionamiento directo*

  La dirección contenida en la instrucción es la dirección de memoria de donde se debe obtener el dato. La instrucción:

  $$\texttt{CAR}\ \ AX,\ \ [500]$$

carga en el acumulador el contenido de la dirección de memoria 500. Si la dirección es la absoluta se denomina *direccionamiento absoluto*.

- *Direccionamiento relativo*

  En el caso del *direccionamiento relativo*, la instrucción no contiene una dirección sino un valor de desplazamiento que se aplica a una dirección de referencia contenida normalmente en un registro, en una pila o en el contador de programa. La instrucción:

  $$\texttt{CAR}\ \ AX,\ \ [BX+4]$$

  carga en el acumulador el contenido de la dirección obtenida al sumar 4 al contenido del registro BX.

- *Direccionamiento indirecto*

  En el direccionamiento indirecto la dirección contenida en la instrucción contiene la dirección en que se encuentra el operando. La instrucción:

  $$\texttt{CARI}\ \ AX,\ \ [500]$$

  carga en el acumulador el contenido de la posición de memoria cuya dirección se encuentra en la dirección 500.

- *Direccionamiento implícito*

  En este caso se hace referencia a un registro que, por quedar sobreentendido, no se menciona en la instrucción.

  En nuestro ejemplo de lenguaje máquina (ver 3.3.2) se sobreentiende que las operaciones se realizan y almacenan en el registro acumulador. En consecuencia, el direccionamiento de este registro queda implícito.

El siguiente cuadro resume los tipos de direccionamiento descritos:

- Direccionamiento
  - Inmediato
  - Directo
  - Absoluto
  - Relativo
    - al contador de programa
    - a un registro
    - a pila
  - Indirecto
  - Implícito (con acumulador)

3.4.2 Subrutinas

Pilas

Las pilas son estructuras secuenciales de datos cuyo acceso se realiza por un extremo. Es semejante a la acción de apilar una serie de libros sobre una mesa: se puede poner un libro sobre la pila o quitar el del extremo superior o cima, pero no insertar ni sacar libros intermedios; para acceder a uno de los libros de la parte inferior hay que retirar previamente los anteriores. Se dice que estas estructuras son de tipo *LIFO* (*Last In First Out*: último en entrar primero en salir).

La mayoría de los procesadores disponen de una pila para almacenar ordenadamente ciertos valores y operaciones. Sobre una pila se definen dos operaciones básicas: meter y sacar datos. Para su funcionamiento se utiliza un puntero que señala a la cima de la pila. Para meter un dato en la pila se decrementa el valor del puntero para que apunte a una posición sobre la actual, y a continuación, se transfiere el dato a esta posición. Para sacar el dato, se envía su valor y después se incrementa el valor del puntero. De esta forma la posición anteriormente ocupada queda liberada.

Estas operaciones suelen expresarse por sus mnemotécnicos en inglés: `PUSH` d y `POP` d. La primera introduce el valor contenido en la dirección d en la pila, y la segunda saca el valor del extremo de la pila y lo deposita en la dirección d.

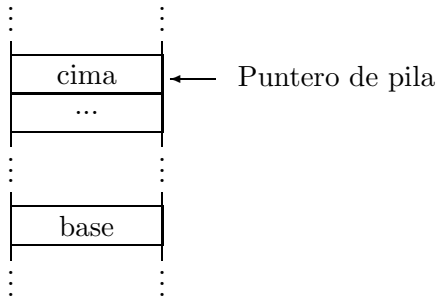


Figura 3.3. Una pila y el puntero de pila.

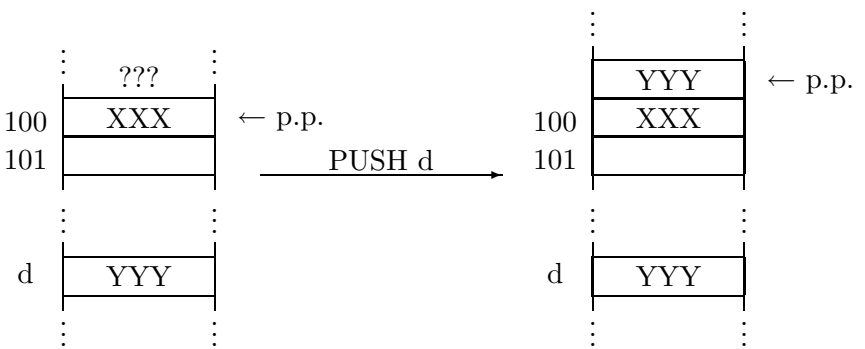


Figura 3.4. Interpretación gráfica de la orden `PUSH`.

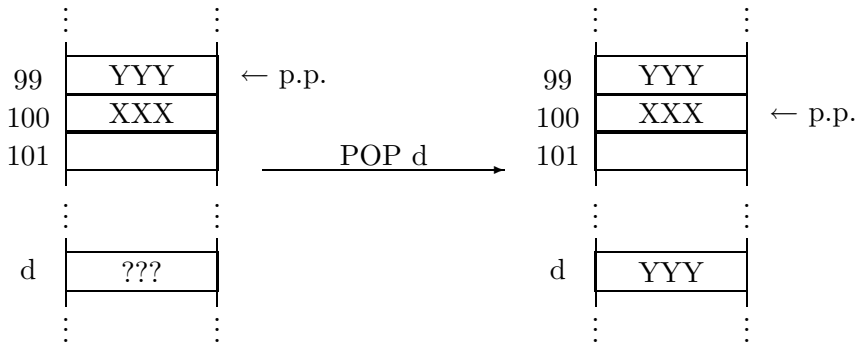


Figura 3.5. Interpretación gráfica de la orden `POP`.

Llamadas a subrutinas

Un subprograma es un conjunto de instrucciones que realizan una acción concreta que se puede repetir varias veces a lo largo del programa, como hemos visto en el ejemplo del apartado 3.3.4. En vez de repetir la codificación de las instrucciones cada vez que se necesiten, éstas se codifican una única vez, formando un *subprograma* o *subrutina*, y se le llama desde los distintos puntos del programa. Los lenguajes de máquina disponen de instrucciones para realizar las llamadas a subprograma, consiguiéndose así una estructura modular con un programa principal que llama a sus respectivos módulos cuando lo necesita.

La instrucción de llamada va acompañada de la dirección de entrada del subprograma:

`CALL` d

Esta instrucción llama al subprograma situado en la dirección d. Para ello, la carga en el registro contador de programa, en forma semejante a las instrucciones de salto `COND` o `GOTO`.

Sin embargo, una vez que el subprograma haya sido ejecutado, hay que volver al programa principal, por lo que se hace necesario conocer

y almacenar la dirección de retorno, normalmente la dirección siguiente a la de la llamada al subprograma, en el programa principal. Esta dirección se suele almacenar en la pila.

En consecuencia, antes de producirse un salto a un subprograma, la instrucción de llamada debe almacenar la dirección de retorno al programa principal, que es la dirección siguiente a la de llamada, en la pila. Cuando se llega a la última instrucción del subprograma el procesador saca la dirección de retorno de la pila y la carga en el registro contador de programa, con lo cual prosigue la ejecución del programa principal.

Los subprogramas deben terminar en una instrucción que indique al procesador que debe volver al programa principal. Suele expresarse con el mnemotécnico:

```
RET
```

que indica el RETorno desde un subprograma.

3.4.3 Interrupciones

El trabajo del procesador puede verse interrumpido por distintos motivos; operaciones de E/S, errores, paradas para depuración, etc.

Una *interrupción* del procesador es similar a lo que sucede cuando estamos leyendo un libro y suena el teléfono: el dispositivo externo, el teléfono, avisa de su disposición a realizar una operación de E/S; el procesador, el lector, almacena su estado de proceso (se fija en la página y el lugar de la página donde se encuentra leyendo), y pasa a ejecutar un subprograma de servicio de la llamada telefónica (responde al teléfono, toma notas, etc., y por último cuelga). A continuación el procesador reanuda su estado anterior a la llamada y prosigue con su proceso (el lector toma el libro, busca la página y continúa con su lectura).

Las interrupciones pueden programarse (interrupciones de software) mediante las correspondientes instrucciones de máquina, o pueden producirse por los controladores del sistema y de dispositivos, a través del bus de control (interrupciones de hardware).

Se puede establecer una jerarquía de interrupciones dependiente de su prioridad; ciertas interrupciones deben ser atendidas de inmediato, incluso deteniendo otras interrupciones en curso de ejecución.

Cada tipo de interrupción determina la llamada a un subprograma de servicio, que contiene las instrucciones necesarias para atender la interrupción. Antes de atender la interrupción, el procesador almacena el contenido de los registros, generalmente en una estructura de tipo pila. A continuación se ejecuta el subprograma, de forma similar a las llamadas de subprogramas (subrutinas). Una vez atendida la interrupción, el procesador recupera los contenidos de sus registros de la pila y prosigue la ejecución del programa principal.

Para el tratamiento de las interrupciones los lenguajes de máquina suelen disponer de instrucciones específicas, que en forma nemotécnica suelen expresarse como:

```
INT   t
IRET
```

La primera genera una `INT`errupción de tipo t, y entre otras acciones llama al subprograma de servicio t. La segunda marca el fin de dicho subprograma y provoca el retorno de la interrupción.

3.5 Otras arquitecturas

En los últimos años se ha introducido una gran variedad de nuevas arquitecturas de computadores, especialmente orientados al *procesamiento en paralelo*, esto es, con capacidad para realizar varias operaciones simultáneamente. Este modo de funcionamiento contrasta con el *procesamiento secuencial*, en que las instrucciones se ejecutan una tras otra.

Actualmente, la mayoría de los computadores incorpora algunas características paralelas a bajo nivel sin que por eso puedan ser llamados con propiedad computadores paralelos. Éstos disponen en general de varios procesadores o Unidades de Procesamiento (UP).

En esta sección se presentan, muy brevemente, algunos esquemas básicos que siguen las distintas arquitecturas paralelas existentes en la

actualidad. Principalmente, estos modelos se diferencian entre sí en que las distintas UP pueden estar dotadas con una memoria local o no, compartir o no una memoria y operar sincrónicamente o no. De modo muy general los computadores paralelos pueden clasificarse en alguna de las tres clases siguientes:

1. *Arquitecturas sincrónicas.* Este tipo de computadores se caracteriza por realizar paralelamente operaciones coordinadas por un reloj, una UCP o un controlador vectorial globales; a su vez pueden clasificarse en algunos de los tipos siguientes:

   - *Procesadores Vectoriales.* Caracterizados por tener varias unidades aritmético-lógicas encadenadas, que permiten realizar cálculos aritméticos y lógicos tanto con vectores como con escalares.
   - *Arquitecturas SIMD.* Esta denominación (del inglés *Single Instruction Multiple Data*) caracteriza a aquellos computadores con una unidad central, varios procesadores y una red que permite la comunicación entre procesadores e intercambio de datos con la memoria. Esta red de interconexión permite que el resultado obtenido por un procesador sea comunicado a otro procesador que lo necesite como argumento.

2. *Arquitecturas MIMD.* Este tipo de arquitecturas (del inglés *Multiple Instructions Multiple Data*) emplea varios procesadores que pueden ejecutar, de forma asíncrona, programas independientes que usan datos locales. Por lo tanto, los computadores MIMD son especialmente útiles cuando el paralelismo de la solución buscada requiere que los procesadores trabajen de manera esencialmente autónoma.

   Las arquitecturas MIMD pueden clasificarse, a su vez, dependiendo del modelo de memoria que utilicen.

   - *De Memoria Compartida.* Las arquitecturas de memoria compartida consiguen la coordinación entre los distintos procesadores mediante una memoria global, compartida. Es interesante observar que en este tipo de computadores se unen

varios *procesadores* de propósito general que comparten una memoria global, en lugar de varias UCPs con su propia gestión de periféricos de entrada y salida.

- *De Memoria Distribuida.* Las arquitecturas de memoria distribuida conectan los *nodos* (un procesador autónomo junto con su memoria local) mediante una red de interconexión entre los procesadores. Los nodos comparten datos explícitamente *pasándose* mensajes a través de la red de interconexión, ya que no hay memoria compartida.
  Se han propuesto varias *topologías* de interconexión para arquitecturas de memoria distribuida. Entre ellas se encuentran las topologías en *anillo*, en *red*, en *árbol* y las topologías *hipercúbicas*.

3. *Arquitecturas basadas en el paradigma MIMD.* Por último, podemos encontrar un tercer grupo de arquitecturas que no encajan en los dos grandes grupos anteriores. Esta clase de computadores están basados en el principio de asincronía y manipulación paralela de múltiples instrucciones y datos; sin embargo, cada una de las siguientes arquitecturas tienen alguna característica propia que la separa de una máquina MIMD.

   - *Híbridos SIMD-MIMD.* Ésta es una arquitectura experimental en la que se permite que partes de una arquitectura MIMD puedan ser controladas de modo SIMD.
   - *De Flujo de Datos.* La característica fundamental de las arquitecturas de flujo de datos es su paradigma de ejecución, en el que una instrucción se ejecuta tan pronto como sus operandos están disponibles. De este modo, la secuencia de instrucciones ejecutadas está basada en la dependencia de los datos, permitiéndose así explotar la concurrencia en los niveles de tarea, rutina e instrucción.
   - *Dirigidas por la demanda* (*demand-driven*). Este tipo de arquitecturas, también llamadas de *reducción*, utilizan un paradigma de ejecución en el que una instrucción se manda ejecutar sólo cuando sus resultados se necesitan como operandos

para otra instrucción que se está ejecutando; este paradigma se conoce también como evaluación perezosa (del inglés *lazy*), ya que se ejecutan sólo las instrucciones estrictamente necesarias para la evaluación pedida.

3.6 Ejercicios

1. Determine el número de líneas de los buses de datos y direcciones precisos en un ordenador con:
   (a) una memoria de 64 K palabras de 1 byte, y
   (b) una memoria de 16 Mb, en palabras de 2 bytes.
2. Dado un tamaño de palabra de memoria de 2 bytes y un bus de direcciones de 20 líneas, calcule el tamaño de memoria direccionable.
3. A partir de los datos de un modelo de computador concreto, extraiga los valores de las siguientes magnitudes, o indique sus características:
   (a) Tamaño de la memoria principal.
   (b) Número de líneas del bus de direcciones y de datos.
   (c) Tipos de memorias utilizadas (ROM, RAM, ...), y su velocidad de acceso.
   (d) Tamaño de memoria caché.
   (e) Velocidad de reloj.
   (f) Tipo de coprocesador.
   (g) Número de controladores en serie y en paralelo.
   (h) Unidades de disco: tipos, capacidad y velocidad de acceso.
   (i) Otros periféricos disponibles.
4. Desarrolle un programa en lenguaje de máquina (usando instrucciones mnemotécnicas) para sumar los 10 primeros números naturales.
5. Desarrolle un programa en lenguaje máquina que copie el contenido de las posiciones 1000 a 1010 de memoria en las 2000 a 2010.
6. Amplíe el juego de instrucciones con `POP` y `PUSH` para desarrollar un programa en lenguaje máquina que copie el contenido de las direcciones de memoria 1000 a 1010 en la pila, y desde la pila en las direcciones 2000 a 2010.

7. Amplíe el juego de instrucciones con `CALL` y `RET` para desarrollar un un subprograma en lenguaje máquina que, al ser llamado, sustituya el valor almacenado en el acumulador por su opuesto.
8. Con el juego de instrucciones del ejercicio anterior, escriba un subprograma en lenguaje máquina que, al ser llamado, sustituya los valores almacenados en las posiciones de memoria 1000 a 1010 por sus opuestos.
9. Escriba un programa en lenguaje máquina que genere un ciclo infinito.
10. Escriba un programa en lenguaje máquina que escriba copias de sí mismo en la memoria disponible.

3.7 Comentarios bibliográficos

- Este capítulo resume, enormemente, un área vastísima de la informática. Por consiguiente, sólo puede constituir una introducción sin pretensiones de algunos conceptos básicos útiles sobre la estructura física de los computadores digitales, y de la coordinación entre éstos y el soporte lógico, al más bajo nivel, que es el de la máquina.
- En [PLT89] puede encontrarse un enfoque muy didáctico de este tema, presentado junto con la descripción de un modelo real, el ODE (Ordenador Didáctico Elemental), desarrollado por los autores.
- El texto [Mei73] es una obra clásica sobre los contenidos de este capítulo, que ha servido para la formación de muchas generaciones de técnicos en computación, por lo que su referencia es obligada. Se advierte, sin embargo, que no se trata de un mero texto introductorio, aunque su clara presentación facilita un buen número de conceptos que, por sí mismos, no son sencillos.
- En [MW84] puede verse cómo se concretan muchos de los contenidos presentados en este tema en dos procesadores concretos, el 8088 y el 8086.
- En [FM87] se presenta brevemente una visión panorámica bastante amena sobre las nuevas arquitecturas de computadores. Un enfoque más técnico puede hallarse nuevamente en el último capítulo de [Mei73].

Capítulo 4

Sistemas Operativos

Un computador sin *software* es algo sencillamente inútil salvo, quizá, como elemento decorativo. Un sistema operativo constituye, probablemente, la parte más importante del conjunto de *software* que acompaña a cualquier computador moderno. El sistema operativo controla todos los recursos del computador y ofrece la base sobre la cual pueden escribirse los programas de aplicación.

El objetivo de este capítulo consiste en exponer los elementos básicos de un sistema operativo, presentar sus funciones fundamentales y establecer las cualidades que cabe esperar de él.

En las primeras secciones se definen los conceptos básicos acerca de un sistema operativo y las funciones que debe realizar. Posteriormente se establece una clasificación, no exhaustiva, de los sistemas operativos según sus características más sobresalientes.

4.1 Cometido de un sistema operativo

4.1.1 Funciones de los sistemas operativos

En el capítulo anterior se ha mostrado someramente la estructura interna de un computador, con su(s) procesador(es), sus memorias principal y secundaria, y los periféricos. Si todo programador de aplicaciones tuviera que controlar "a mano" todos los componentes de la estructura interna del computador cuando hace sus programas, muy probablemente, no existirían tantos programas en el mercado. Por poner un ejemplo, emplear la instrucción ESCRIBIR ARCHIVO es mucho más simple que tener que preocuparse por mover las cabezas lectoras del disco hasta una posición adecuada, esperar que se estabilicen, dirigir la información desde la memoria e ir escribiendo la información en el disco.

Conviene al programador de aplicaciones (o al mismo usuario) desentenderse de los complejos detalles del hardware. La idea fundamental para conseguirlo consiste en ocultar los escabrosos detalles del hardware puro con software (el *sistema operativo*) especialmente diseñado para permitir un uso más fácil y racional de todas las partes del sistema.

Del mismo modo, el sistema operativo presenta un interfaz simple entre el programador o usuario y otros aspectos del hardware, como son el manejo de interrupciones, relojes, gestión de memoria y otras características de bajo nivel.

Según este enfoque, el sistema operativo tiene la misión de presentar al usuario una *máquina virtual* que sea más fácil de programar que el hardware puro y, por lo tanto, aumente la efectividad del computador al evitar la necesidad de trabajar a bajo nivel. Por consiguiente, puede considerarse un sistema operativo como un interfaz adecuado entre el usuario y el hardware.

Un sistema operativo también tiene la función de controlar y administrar de forma ordenada el uso de todos los recursos del computador. Esta función se hace especialmente necesaria cuando un computador está siendo compartido por varios usuarios simultáneamente; en este caso, la necesidad de distribuir convenientemente el tiempo de trabajo de la

UCP, los espacios de memoria y los periféricos es evidente. Desde este punto de vista, la tarea del sistema operativo es llevar el control de quién está utilizando cada recurso y dirimir los posibles conflictos entre varios procesos o usuarios que intenten acceder a la vez a un mismo recurso.

No existe acuerdo entre los diferentes textos al señalar cuáles son las funciones de un sistema operativo, ya que este concepto ha evolucionado con su desarrollo. En general, un sistema operativo debe poder desempeñar al menos las siguientes funciones:

- Facilitar la comunicación hombre-máquina.
- Gestionar los recursos: procesador(es), memoria(s) y periféricos, facilitando su manejo al usuario.
- Gestionar la información (los *archivos*) contenida en los periféricos de almacenamiento y la organización de esa información (en *directorios*).
- Controlar la ejecución de aplicaciones.

4.1.2 Formas de trabajo de los sistemas operativos

Los sistemas operativos han evolucionado para cubrir las distintas necesidades de proceso sobre máquinas de tamaño y potencia muy diversos. Aparecen así diferentes formas de trabajo y sistemas operativos especializados en cada una de ellas.

Una de las principales distinciones se da entre los sistemas operativos de *propósito general* y los *dedicados*.

Los primeros, que son los más utilizados, se diseñan con una intención de polivalencia, de forma que sobre ellos puedan ejecutarse programas diversos: editores, compiladores y aplicaciones de todo tipo. Dentro de los sistemas operativos de propósito general puede distinguirse entre *monousuario* y *multiusuario*, dependiendo de si pueden atender o no a más de un proceso al mismo tiempo. Claramente, los sistemas multiusuario son más complejos pues, además de desempeñar todas las funciones propias de uno monousuario, deben atender a otros aspectos, tales como el

reparto del tiempo de trabajo del (los) procesador(es) entre los distintos usuarios, la separación entre los datos de los mismos, etc.

En los sistemas operativos de propósito general, hay dos formas de trabajo básicas: el modo *interactivo* y el *proceso por lotes*. En el primero, el usuario tiene acceso a los recursos del sistema a través de su terminal (local o remoto), estableciendo un diálogo con el computador, donde las órdenes y las respuestas tienen lugar de un modo casi inmediato.

En un proceso por lotes, los diferentes trabajos junto con sus datos se suceden entre sí, de forma que su ejecución se realice automáticamente, sin ningún tipo de comunicación entre el usuario y el sistema, hasta la finalización del trabajo.

Dentro de los sistemas operativos dedicados, se suele distinguir entre los de control de procesos, de consulta de bases de datos y transaccionales. En el *control de procesos*, el computador recibe continuamente datos (realimentación) sobre el sistema bajo control, reaccionando en consecuencia. En la mayoría de estos procesos, la variable tiempo tiene una importancia relevante; por ejemplo, en un horno industrial, un retraso en la reducción de la temperatura puede tener funestas consecuencias. En estos procesos, se dice que el sistema debe operar en *tiempo real*.

Los *sistemas de consulta* deben permitir al usuario el acceso a una o varias bases de datos de una forma cómoda, sin tener que conocer los detalles internos de su organización y funcionamiento. de su estructura. Se puede citar como ejemplo un sistema de consultas de historias clínicas en un hospital.

Por último, los *sistemas transaccionales* permiten la realización de operaciones sobre el contenido de bases de datos. Se han extendido enormemente en los últimos años. Por ejemplo, así se llevan a cabo los movimientos de cuentas en el sector bancario, la reserva y venta de billetes en agencias de viajes, etc.

Estos sistemas deben garantizar la opacidad de la información confidencial, actualizar rápidamente la base de datos y evitar ciertas operaciones simultáneas sobre los mismos datos (por ejemplo, dos personas que reservan el mismo asiento).

4.2 Conceptos básicos de los sistemas operativos

4.2.1 Procesos

Es importante el concepto de proceso para entender cómo funcionan los sistemas operativos. Un *proceso* consiste, esencialmente, en una rutina de acciones del sistema que puede ser ejecutada independientemente de otro proceso e, incluso, en paralelo. Un programa de usuario en ejecución puede considerarse un proceso, aunque un proceso puede involucrar la ejecución de más de un programa y, recíprocamente, un programa o rutina de instrucciones puede estar involucrado en más de un proceso. Por esta razón es más útil el concepto de proceso que el de programa al referirnos al sistema operativo.

Todo proceso creado puede ser tratado como un seudorrecurso pues, al contrario que los recursos reales, los procesos son entidades efímeras que desaparecen una vez completada su misión. Podría argüirse que los procesos son consumidores de recursos en lugar de ser recursos por sí mismos; sin embargo, una vez que han entrado en el estado transitorio de ser procesos, pueden ser "manejados" por otros procesos.

La gestión de procesos es una tarea fundamental de los sistemas operativos; concretamente, la mayoría de los sistemas operativos deben soportar la ejecución *concurrente* de procesos y resolver los problemas que ésta plantea, ofreciendo medios para la *sincronización* de los procesos, controlando su *exclusión mutua* respecto de ciertos recursos, y evitando, en la medida de lo posible, las situaciones de *bloqueo*. Estos conceptos se explican a continuación.

Dos o más procesos se dicen *concurrentes* cuando se están ejecutando simultáneamente. Cuando existen más procesadores que procesos, cada proceso se ejecutará sin conflicto en un procesador (paralelismo); si, por el contrario, existen más procesos que procesadores puede obtenerse una "simultaneidad aparente" (concurrencia) haciendo que los procesadores alternen cortos períodos de tiempo de dedicación a los distintos procesos.

Si dos procesos concurrentes utilizan recursos distintos podrán con-

vivir sin interferencias pero si, por el contrario, los procesos necesitan los mismos recursos puede surgir conflicto. Supóngase que ambos recursos pretenden acceder a una impresora: es absolutamente inviable conmutar la dedicación de la impresora entre ambos procesos ya que, en ese caso, el resultado sería caótico. Este es un caso típico de necesidad de *exclusión mutua* de dos procesos respecto a un recurso.

Dos procesos concurrentes pueden necesitar cooperar al realizar sus tareas; por ejemplo, un proceso que genera líneas de texto que envía a un *buffer* y otro que las consume, tomándolas del *buffer* de datos y enviándolas a la impresora. Cuando así ocurre, es necesario que exista una *sincronización* de ambos procesos.

Por último, puede darse la circunstancia de que dos (o más) procesos en ejecución concurrente se *bloqueen* mutuamente, en el sentido de que cada uno de ellos necesita un recurso que el otro posee. Llegados a este punto la ejecución se detiene, ya que cada proceso espera que el otro libere la información (o el recurso) que necesita. Esta situación se llama *bloqueo* (en inglés *deadlock*).

4.2.2 Archivos

La información es almacenada por el ordenador en dispositivos, como por ejemplo los discos. Para que el usuario pueda ignorar la estructura física del disco, el sistema operativo debe presentar la información almacenada de una forma organizada, que permita acceder a ella de una manera lógica y sencilla.

En este contexto, se entenderá por *archivo* un conjunto de información relacionada (definida por el usuario o no). En él pueden estar contenidos datos o programas. Un sistema operativo debe gestionar los archivos de una forma eficiente; para ello son necesarias operaciones tales como la creación, modificación y borrado de archivos. El sistema operativo debe proporcionar estas operaciones, permitiendo al usuario referirse a los archivos mediante nombres simbólicos.

Al manipular los archivos, el computador puede acceder a ellos de dos modos, dependiendo del medio de almacenamiento en que reside el

archivo (véase 3.1.3): el acceso secuencial y el acceso directo.

- Con *acceso secuencial* se recorren los elementos de un archivo uno tras otro, consecutivamente. Este método de acceso viene impuesto por ciertos medios de almacenamiento, tales como las cintas magnéticas, aunque también puede darse en otros dispositivos, como los discos magnéticos, cuando los archivos tienen esta organización.

- Con *acceso directo*, los diferentes elementos pueden localizarse directamente por su posición, sin necesidad de recorrer los anteriores para llegar hasta ellos. Éste es el modo de acceso típico de los discos magnéticos.

Cuando el número de archivos crece, resulta incómoda su localización y manipulación. Para facilitarla, el sistema operativo presenta al usuario la posibilidad de agruparlos en compartimentos, llamados *directorios*. El sistema operativo deberá manejar la organización de directorios, ocultando al usuario los detalles particulares de la localización física de los archivos y directorios, facilitándole en cambio operaciones para gestionar cómodamente el sistema de directorios.

La técnica más comúnmente utilizada es usar una *estructura en árbol*, en la que existe un *directorio raíz* que puede contener archivos y/o (sub)directorios; a su vez cada directorio puede contener más archivos y/o más (sub)directorios. La figura 4.1 muestra un ejemplo de sistema de archivos.

Una imagen visual bastante útil de un sistema de archivos con estructura de árbol se consigue al considerar un directorio como una carpeta que contiene un cierto número de documentos (los archivos). En principio nada nos impide que coloquemos una carpeta dentro de otra (salvo quizá la limitación de espacio).[1]

Una operación fundamental en la gestión de un sistema de archivos es la búsqueda eficiente de un archivo determinado. Si el número de

[1]Esta es la representación utilizada en los interfaces gráficos de usuario como los de *Apple Macintosh* y *Windows*.

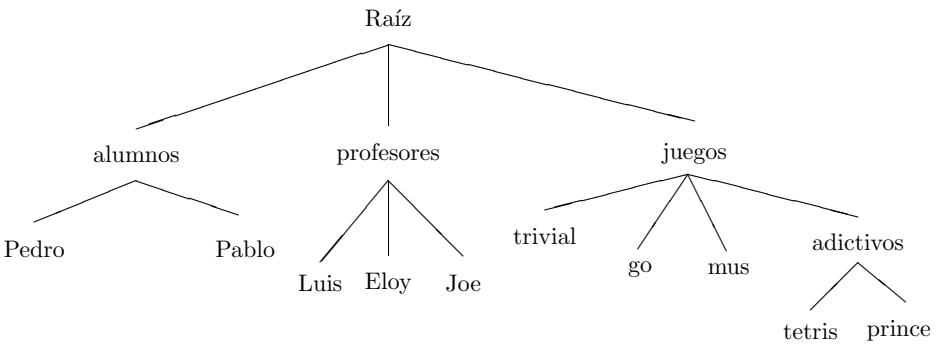


Figura 4.1. Un sistema de archivos con estructura de árbol.

archivos es pequeño lo más simple y lo mejor es buscar secuencialmente archivo tras archivo, pero si el sistema contiene muchos archivos esta táctica llevará mucho tiempo y, como esto es lo habitual, se han ideado estrategias para hacer el acceso más eficiente.

La idea de *directorio de trabajo* permite agilizar la búsqueda de archivos en el sistema; para ello es fundamental la noción de *camino* (en inglés *path*) desde la raíz del árbol hasta una rama determinada; por ejemplo, en el árbol de directorios de la figura 4.1 los juegos adictivos tienen el *path* `\juegos\adictivos`. Si el directorio de trabajo fuera `\juegos` entonces `adictivos\tetris` (camino *relativo* al directorio de trabajo) tiene el mismo significado que `\juegos\adictivos\tetris` (camino *absoluto*).

Para la optimización de la búsqueda de archivos se suele buscar sólo en el directorio de trabajo y en algunos otros *paths* especificados por el usuario.

Otra prestación ofrecida por la mayoría de los grandes sistemas operativos es la *protección* de la información. Debido a la facilidad con que puede borrarse la información almacenada, los sistemas operativos ofrecen dos mecanismos principalmente para asegurar archivos y directorios contra los descuidos, propios o ajenos:

- un sistema de *copias de seguridad* (*backup*), que se pone en mar-

cha automáticamente ante cada modificación efectuada sobre cada archivo.

- un sistema de *atributos*, que permite marcar los archivos que necesitan copia de seguridad, como *imborrables*, como *no susceptibles de modificación*, etc.

Otro aspecto de la protección, igualmente importante en sistemas multiusuario, consiste en permitir el acceso y garantizar que el trabajo de cada usuario es privado, ocultándolo a los demás. Este efecto se consigue por lo general mediante un sistema de marcas, que identifica al propietario de los distintos archivos y directorios, y de *contraseñas*, que permite reconocer el usuario concreto que se encuentra ante cada terminal.

4.2.3 Núcleo

Se emplea frecuentemente el término *núcleo* o *kernel* para referirse a aquella parte del sistema operativo más cercana a la máquina y que, por lo tanto, siempre está almacenada en la memoria principal (RAM). El núcleo es un código muy frecuentemente usado por otros programas de un nivel superior (aplicaciones o utilidades). Por esta razón se presenta, aparentemente, como una ampliación del conjunto de instrucciones de la máquina. El núcleo desempeña, al menos, las siguientes funciones:

1. *Manejo de interrupciones.* Cada sistema operativo tiene un cierto número de tipos posibles de interrupciones, (véase la sección 3.4.3). El diseño del núcleo debe garantizar que, cuando la interrupción sea atendida para permitir que se ejecute otra tarea, la información que se estaba procesando quede correctamente protegida y almacenada, garantizándose que la actividad interrumpida pueda reanudar su ejecución en el punto en que la dejó, cuando sea atendida la interrupción.

2. *Soporte de entrada y salida.* El núcleo debe controlar los avisos de los procesos que necesitan comunicarse con los periféricos e interceder en caso de conflicto.

4.2.4 Multiprogramación

En un sistema de *multiprogramación* el computador puede ejecutar varios programas simultáneamente.

La idea básica es aprovechar la mayor rapidez del procesador respecto de los periféricos, discos o impresoras, en atender una petición, o bien aprovechar el tiempo que tarda un usuario en introducir sus órdenes a través del teclado. Con todo esto, se aprovecha mejor la potencia del procesador que, a fin de cuentas, es el elemento más caro del computador.

Para poder trabajar con multiprogramación, el sistema operativo debe disponer de un mecanismo que permita que la memoria sea asignada correctamente, y compartida entre los distintos trabajos que se ejecutan.

A diferencia de la memoria, en la que pueden caber varios programas a la vez, la UCP en un instante concreto está dedicada solamente a un programa de usuario o bien a realizar una de las múltiples tareas precisas para la gestión del sistema. Hace falta un mecanismo que planifique esta actividad y su diseño es fundamental para el sistema operativo; este mecanismo se conoce como *planificación* (en inglés *scheduling*).

El objetivo de la planificación es optimizar el uso de la UCP. Los trabajos que están esperando turno se ponen "en cola", de la cual se debe extraer el candidato más idóneo para utilizar la UCP en cada momento. El proceso de selección entre los miembros de esta cola es realizado por el planificador, pudiendo existir más de uno, por ejemplo, para controlar los sistemas de tiempo compartido a corto, medio y largo plazo.

Para elegir el mecanismo adecuado hay que tener en cuenta cuál es el uso principal del computador. Si sólo se están ejecutando procesos *por lotes* (véase pág. 104 o la sección A.2.5) durante la noche es posible realizarlos uno tras otro, pero si existen usuarios realizando consultas puede ponerse como objetivo que ninguno de ellos esté más de un determinado número de segundos sin atención. Las disciplinas de gestión de colas van desde la más simple, consistente en una cola en la que el primero que llega es el primero que es atendido, hasta mecanismos muy complejos en máquinas con varios procesadores, donde las decisiones se

toman teniendo en cuenta otros parámetros además del tiempo, como por ejemplo la prioridad (importancia) de cada proceso.

En los sistemas medios tiene interés conocer el mecanismo denominado *round-robin*, que ha sido especialmente diseñado para sistemas de tiempo compartido en los que hay varios usuarios en distintas terminales compartiendo un computador. Cada uno de ellos debe tener la impresión de que el sistema trabaja sólo para él; para ello, se define una unidad de tiempo (llamado *quantum*) que oscila entre 10 y 100 milisegundos, en función del sistema, y la cola se trata como si fuera una circunferencia de forma que el algoritmo va concediendo una parte de tiempo fija a cada usuario, transcurrido el tiempo asignado se le retira el control de la UCP y se cede al siguiente, y así sucesivamente.

4.2.5 Interfaz de usuario

Sea cual fuere el sistema operativo empleado, hay un conjunto de programas que regulan la comunicación de la máquina con el usuario y señalan las reglas que rigen esta comunicación. Esto es lo que se denomina *interfaz de usuario* Hay dos tipos de interfaz:

- *Interfaz de mandatos*, en la que el procedimiento de comunicación del usuario con el computador consiste en la introducción de instrucciones por medio del teclado.

- *Interfaz gráfico*, en la que el procedimiento consiste en la selección por el usuario mediante un periférico adecuado (habitualmente un ratón) entre varias opciones que le son presentadas en la pantalla mediante gráficos, iconos o distintos tipos de esquemas.

Los interfaces de mandatos se emplearon en los primeros computadores y aún en la actualidad se emplean en entornos como DOS y UNIX. Sin embargo, cada vez más se tiende a la utilización de interfaces gráficos por lograrse con ellos una mayor productividad a costa de una menor fatiga del usuario, ya que éste no necesita conocer los comandos precisos para activar cada una de las opciones disponibles.

El sistema pionero del interfaz gráfico ha sido el de los computadores *Apple-Macintosh*. Actualmente existe un potente interfaz gráfico para

el sistema DOS como es WINDOWS. El sistema UNIX también tiene varios sistemas de ventanas como OPEN-LOOK y OPEN-WINDOWS. Recientemente, ha aparecido una versión de WINDOWS independiente de DOS: el sistema operativo WINDOWS-NT, que permite gestionar computadores, impresoras, teléfonos o telefax en una red digital única.

4.2.6 Gestión de la memoria

En teoría, la potencia del procesador suele permitir gran cantidad de operaciones por segundo pero, dado que para que un programa sea ejecutado debe estar en la memoria principal, habrá que gestionar la ocupación de memoria por parte de los programas que el procesador está atendiendo.

Para optimizar la memoria del computador existen dos soluciones extremas: tener una memoria real muy grande (lo cual resulta caro) o tener los datos en los discos y hacerlos entrar y salir todas las veces que haga falta (lo que resulta ineficiente). Entre ambas soluciones existen varias posibilidades:

1. Tener la memoria dividida en un *número fijo de particiones de tamaño constante*, de modo que en cada una de ellas puede existir un programa en ejecución. Al elegir esta estrategia surgen dos problemas:

   (a) *Fragmentación interna*: al situar un programa en una partición, se desperdicia la parte sobrante de ésta.

   (b) *Fragmentación externa*: surge al tratar de situar un programa grande, que no puede entrar en ninguna de las particiones vacías, aunque cabría en la memoria si éstas se pudieran agrupar.

   Una forma obvia de solventar estos problemas es no fijar de antemano ni el número ni el tamaño de las particiones de la memoria, permitiendo así una asignación de recursos más dinámica.

2. *Poder dividir la memoria en un número variable de particiones*, en este caso el sistema operativo debe proporcionar al operador

un sistema que pueda variar el tamaño de las particiones y adaptarlas a los programas que están esperando en ese momento. Esta solución no es perfecta, ya que a medida que los programas terminan su ejecución y salen de la memoria dejan un hueco libre, exactamente en el espacio que ocupaban.

A la hora de cargar un programa para su ejecución puede ocurrir que, incluso habiendo suficiente cantidad de memoria disponible, ésta no sea contigua. Entonces se debe decidir si se carga el programa en el primer agujero disponible que tenga suficiente tamaño, o bien en el más pequeño de los posibles. En todo caso, la existencia de estos huecos tiende hacia un tipo de fragmentación externa que puede ser bastante importante.

Para evitar estos problemas surgen varias soluciones:

(a) *Compactación*, mediante la cual los programas son desplazados para conseguir que todo el espacio libre sea contiguo.

(b) *Compartición*, mediante la que un mismo código puede ser usado por varios programas y, por lo tanto, por varios usuarios.

(c) *Memoria virtual*, que en cierto sentido es la contrapartida de la compactación: en lugar de luchar contra la fragmentación, empujando los programas para que quede espacio contiguo, se permite que los programas puedan cargarse por partes en vez de hacerlo como un bloque monolítico.

Memoria virtual

La técnica de memoria virtual permite iniciar la ejecución de un programa sólo con una parte de éste en la memoria, de manera que se pueda solicitar más memoria en cuanto sea preciso.

La mayor ventaja que tiene este sistema es que permite trabajar con programas mayores que el total de la memoria física disponible. Con los computadores modernos no es frecuente que tengamos programas más grandes que el total de la memoria disponible, por lo cual la introducción

de memoria virtual puede parecer un lujo excesivo. Sin embargo, puede ocurrir que muchas partes de un programa no suelen ejecutarse salvo en muy raras ocasiones, como es el código para el manejo de las condiciones de error, tablas internas o rutinas que tratan sólo casos especiales que afectan a muy pocos de los datos. Con la técnica de memoria virtual bastaría mantener en la memoria el núcleo del programa, consiguiéndose, en cualquier caso, un mejor aprovechamiento de la memoria física, que es cara.

La idea de memoria virtual puede ser implantada en un sistema operativo concreto mediante varias soluciones, siendo las más corrientes las de *paginado* y las de *segmentación*.

Memoria virtual paginada.- Las técnicas de *paginación* consisten en la separación de los conceptos de espacio de direcciones y posiciones de memoria, o lo que es lo mismo, entre memoria lógica y memoria física. Para comprender mejor en qué consiste la técnica de paginado consideraremos el siguiente ejemplo:

Supóngase un computador con un campo de direcciones de 16 bits y una memoria principal de 8Kb, en palabras de un byte. En principio se pueden direccionar (espacio de direccionamiento) $2^{16} = 65536 = 64$K palabras de memoria (de un byte) pero sólo existe espacio físico para $8 \cdot 1024 = 8192$ palabras, para direcciones de memoria desde 0 hasta 8191. Como la memoria lógica es mucho mayor que la realmente disponible habrá que diseñar un método que soporte llamadas, por ejemplo, a la dirección 8195 sin que se genere un desagradable mensaje de error. Una posible solución para evitar los errores cuando se producen llamadas a memoria con direcciones superiores a las realmente disponibles, como la 8195, sería usar *aritmética modular* y considerarla simplemente como la dirección $3 = 8195 - 8192$ (del mismo modo que sabemos que las 15 horas son las 3 de la tarde: $3 = 15 - 12$). Esta alternativa implicará la inclusión de una función de conversión entre direcciones lógicas y direcciones físicas.

En un sistema con memoria virtual paginada se ejecutarían los siguientes pasos cuando se recibe la llamada a la dirección 8195, que no

se corresponde con ninguna dirección física:

1. Se salva el contenido de la memoria principal en la memoria secundaria (generalmente el disco).
2. Se localizan las palabras correspondientes a los siguientes 8Kb de memoria direccionable, esto es, las palabras desde 8192 hasta 16383 en la memoria secundaria.
3. Se cargan estas palabras en memoria principal.
4. Se cambia la función de conversión de direcciones de memoria.
5. La ejecución continúa con normalidad hasta que se vuelva a producir otra llamada a una dirección de memoria fuera del espacio físico.

Memoria virtual segmentada.- En este modelo de memoria virtual se refleja de forma clara la división entre programas y datos. La idea consiste en dividir el espacio de direcciones en *segmentos*, cada uno de los cuales corresponderá a una rutina, un programa o un conjunto de datos; en este caso se estará explotando el concepto de *modularidad* de los programas construidos estructuradamente.

La segmentación es una técnica que organiza el espacio lógico de memoria en segmentos (bloques independientes) de tamaño variable, que se colocan en la memoria física mediante algoritmos de localización de espacio libre. Puesto que cada segmento constituye un espacio de direcciones distinto, diferentes segmentos pueden crecer o encoger de manera independiente, sin que se afecten entre sí; por supuesto, aunque los segmentos pueden crecer, un segmento se puede llenar y, en este caso, se producirá un mensaje de error.

La especificación de una dirección en una memoria segmentada (también llamada bidimensional) debe proporcionar dos componentes: un número de segmento y la dirección dentro del segmento. Cada segmento forma una entidad lógica que el programador puede tener en cuenta para asignarle distintos tipos de protección; por ejemplo, si un segmento contiene un procedimiento, entonces se especificará que es un segmento de

ejecución, prohibiéndose la escritura; si contiene una pila se especificará como de lectura y escritura, pero no de ejecución. Esta práctica es de gran utilidad en la detección de errores de programación.

4.3 Clasificación de los sistemas operativos

El criterio más simple de clasificación es aquél que atiende al tamaño y potencia del hardware que manejan, y divide los sistemas operativos entre los grandes sistemas (*mainframes*), los de computadores de gama media (*minis*) y en los microcomputadores o *micros*. Cuanto más potente, y por lo tanto más cara, sea la máquina a la que va dirigido el sistema, mayor es el interés existente en optimizar los recursos, esforzándose el diseñador en conseguir una gestión de tareas dinámica y eficiente que permita gestionar simultáneamente múltiples programas sin intervención del operador, con independencia de los dispositivos, etc.

Otro importante criterio de clasificación es la división entre sistemas *propietarios* y sistemas *abiertos*, es decir, si el fabricante del hardware dispone de control exclusivo sobre el mismo o bien se trata de un sistema abierto a toda la comunidad industrial.

En computadores pequeños y medios es ya una norma la utilización de sistemas abiertos, como UNIX, frente a los sistemas propietarios, caracterizados por el control que sobre ellos ejerce un fabricante de hardware, por ejemplo, el sistema utilizado por los computadores *Apple-Macintosh* o algunos derivados de UNIX como AIX o ULTRIX.

La ventaja de un sistema abierto es la independencia de la máquina sobre la que se instala, por lo que tanto los programas fuente como las bases de datos o las transacciones, pueden ser transferidas de un hardware suministrado por un fabricante a otro distinto, independizando al usuario de los proveedores.

Las ventajas de un sistema propietario es haber sido diseñado para un hardware específico, y por lo tanto obtiene de él las mejores prestaciones posibles. Las interfaces a las bases de datos, monitores de teleproceso y otros productos del fabricante están exhaustivamente probados, por

lo que (al menos en teoría) existirán menos problemas para utilizar tal sistema.

4.4 Ejercicios

1. Haga un resumen de aquellos mandatos que se utilicen en la gestión de archivos y directorios en un sistema operativo concreto (por ejemplo DOS o UNIX).
2. Dado que las tareas de impresión consumen bastante tiempo, es frecuente imprimir en un "segundo plano", y al mismo tiempo realizar otras tareas, siendo ésta una de las aplicaciones típicas de la multiprogramación en sistemas monousuario. Busque los mandatos necesarios para aplicar esta técnica en DOS, y estudie la gestión de las colas de impresión.
3. Acceda a un interfaz gráfico de usuario (por ejemplo WINDOWS) y relacione los mandatos del administrador de archivos con sus equivalentes en el DOS, analizando las ventajas e inconvenientes de cada forma de operar.
4. Estudie el tamaño máximo de programa que se puede cargar y ejecutar en la memoria en un computador bajo DOS utilizando el mandato `MEM`.
5. Diferencie entre los conceptos de disco virtual y de memoria virtual. (Consúltese el apéndice A.)
6. Estudie los mecanismos de protección en un sistema multiusuario concreto, como UNIX. ¿Cuáles de ellos tienen sentido en un sistema monousuario y cuáles no?
7. Estudie los comandos que permiten controlar la ejecución de trabajos en un sistema multitarea como UNIX.

4.5 Comentarios bibliográficos

En este capítulo no se ha hecho más que introducir algunos de los conceptos básicos de los sistemas operativos, sus funciones y sus principales modos de funcionamiento. Para un estudio más profundo, se pueden consultar [Lis86], [Mil89], [PS91] y [Dei93].

En el apéndice se incluye una introducción práctica y rápida, aunque necesariamente incompleta, de dos de los sistemas operativos más conocidos y difundidos en la actualidad. Aunque para un primer contacto con el DOS bastará en general con esta introducción, los manuales del usuario y de referencia

que se distribuyen al comprar un PC lo mantendrán al día sobre las sucesivas actualizaciones del DOS, proporcionándole una información mucho más completa. Para profundizar más en este sistema, un libro muy detallado sobre el DOS es [Jam90].

La bibliografía sobre UNIX es muy extensa, desde introducciones muy generales hasta libros sobre parcelas muy especializadas. Una buena referencia que abarca casi todo el sistema UNIX la constituye [RRF91]. Por otra parte, en [KP87] se dispone de una breve presentación del sistema, con el objetivo de introducir los requisitos necesarios para abordar una muy interesante sección de desarrollo de aplicaciones.

Capítulo 5

Lenguajes de programación

Un *lenguaje de programación* es un lenguaje artificial, diseñado para representar expresiones e instrucciones de forma inteligible para las computadoras.

Los lenguajes de programación son, en gran medida, comparables a los lenguajes naturales: sus símbolos básicos constituyen su *alfabeto*, y con ellos se construye el *vocabulario* del lenguaje, cuyos elementos se llaman *tokens*. Estos tokens se combinan de acuerdo con las *reglas sintácticas* del lenguaje, formando expresiones y sentencias cuyo significado viene dado por la *semántica* del lenguaje.

Los lenguajes de programación son sin embargo considerablemente más simples que los naturales en su sintaxis y, especialmente, en su semántica.

En este capítulo se introducen distintos aspectos relacionados con los lenguajes de programación. En primer lugar, hacemos una presentación evolutiva, mostrando las aportaciones hechas por los lenguajes

dirección			contenido			
...			...			
0000	0000	0010	0010	0000	0000	1101
0000	0000	0011	0000	0000	0000	1100
0000	0000	0100	0101	0000	0000	1101
0000	0000	0101	1100	0000	0000	1010
0000	0000	0110	0000	0000	0000	1101
0000	0000	0111	0101	0000	0000	1100
0000	0000	1000	0001	0000	0000	1101
0000	0000	1001	1101	0000	0000	0011
0000	0000	1010	0011	0000	0000	1101
0000	0000	1011	1110	0000	0000	????
0000	0000	1100	0000	0000	0000	1010
????	????	1101	????	????	????	????
...			...			

Figura 5.1. Expresión de un programa en código máquina.

ensambladores y los de alto nivel y muy alto nivel, fuertemente orientados al problema. La descripción de los lenguajes de programación se efectúa, cada vez en mayor medida, a través de procedimientos formales, evitándose toda ambigüedad y facilitando el estudio de las propiedades de los programas, por lo que estudiamos algunos de esos procedimientos. Finalmente, presentamos los programas procesadores de lenguajes, su función, sus tipos y los entornos de programación, muy evolucionados, que facilitan hoy día las distintas fases de desarrollo de programas.

5.1 Lenguajes de bajo y alto nivel

5.1.1 Lenguajes orientados a la máquina

Hablando estrictamente, los únicos símbolos que comprenden los computadores son dos: el cero y el uno. Combinándolos apropiadamente, se forman las instrucciones, como se ha descrito en el capítulo 3.

La interpretación de estas instrucciones está dirigida a componentes físicos de la máquina subyacente, tales como el registro acumulador o

dir.	contenido
	...
	IN n
	CAR d
m1	RES n
	COND m2
	CAR n
	RES d
	ALM n
	GOTO m1
m2	OUT n
	END
d	10
n	?
	...

Figura 5.2. Un programa en lenguaje ensamblador

las direcciones de los operandos. Por estas razones tan obvias, estos rudimentarios lenguajes se llaman *lenguajes orientados a la máquina* o, más brevemente, lenguajes de máquina.

El programa de la figura 5.1 está escrito en el lenguaje máquina definido en el capítulo 3.

Resulta evidente que, aun siendo los lenguajes máquina el medio de expresión natural para las computadoras, son ciertamente aparatosos para el programador humano que deba redactarlos, leer o modificar los programas escritos en ellos. Presentan además el inconveniente de *no* ser *transportables*, esto es, no ser comprendidos por máquinas diferentes. Los lenguajes evolucionados han surgido para superar estas deficiencias.

El primer paso para hacer más humana la programación consistió en introducir el uso de identificadores nemotécnicos para referir instrucciones y nombrar las direcciones de los operandos mediante símbolos en lugar de hacerlo mediante sus códigos binarios. Nacen así los primeros lenguajes *simbólicos*. El programa de la figura 5.1 podría traducirse a

un lenguaje simbólico como se indica en la figura 5.2.

Aunque el paso dado es importante, los lenguajes simbólicos consisten sólo en una representación más inteligible de las instrucciones de máquina. El siguiente paso consistió en incorporar en los lenguajes simbólicos *macro*instrucciones, para representar secuencias de instrucciones de utilización frecuente. Estos lenguajes se conocen como *ensambladores*.

Enseguida se comprendió la necesidad de escribir programas que tradujeran textos escritos en ensamblador a textos escritos en lenguaje máquina. Estos programas traductores se llaman también ensambladores.

5.1.2 Lenguajes de alto nivel

A diferencia de los lenguajes orientados a la máquina, los lenguajes de alto nivel se componen de "términos" y "frases" relacionados con el problema en resolución. Así por ejemplo, una ecuación de segundo grado $ax^2 + bx + c = 0$ se podría resolver en un lenguaje de alto nivel como se indica en la figura 5.3.

Por razones obvias, se dice que los lenguajes de alto nivel están orientados al problema. Esta ventaja es doble: por un lado, permiten al programador concentrar su atención sobre el problema, ignorando los detalles propios de la máquina concreta que lo ha de resolver; por otra parte, como el discurso de los programas de alto nivel es independiente de la máquina, son altamente compatibles (transportables) entre máquinas diferentes.

En la figura 5.4 se clasifican los lenguajes de programación, de más evolucionados a más cercanos a la máquina.

En los últimos años se ha desarrollado una generación de lenguajes de muy alto nivel, que incorporan diversos mecanismos para aumentar su capacidad expresiva en algún aspecto. En el apartado siguiente introducimos algunos de esos modelos.

```
...
readln (a, b, c);
discrim := sqr(b) - 4 * a * c;
if discrim >= 0
    then  begin
              writeln ((-b+sqrt(discrim))/(2*a));
              writeln ((-b-sqrt(discrim))/(2*a))
          end
    else  begin
              real := -b/(2*a);
              imag := sqrt(-discr)/(2*a);
              writeln(real, '+', imag, 'i');
              writeln(real, '-', imag, 'i');
          end
...
```

Figura 5.3. Fragmento de un programa en un lenguaje de alto nivel

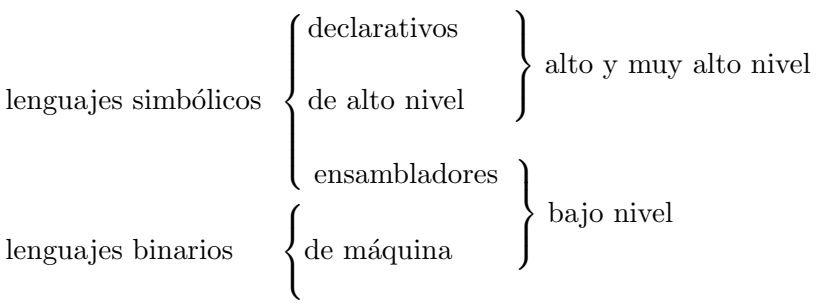


Figura 5.4. Distintos tipos de lenguajes de programación.

5.1.3 Paradigmas de programación

Programación imperativa—programación declarativa

La forma tradicional de programar recibe el nombre de *programación imperativa*, ya que un programa no es más que una secuencia de órdenes o instrucciones que debían ser ejecutados por el computador. Este modelo de computación ha influido fuertemente en la naturaleza de los lenguajes de programación; aunque la evolución de éstos ha permitido esconder muchas características de bajo nivel, los lenguajes convencionales todavía presentan un estilo de programación basado en dotar al computador de procedimientos para resolver problemas, diciéndole *cómo* se resuelven. Este paradigma de programación obliga al programador a tener siempre presentes los detalles particulares del computador que ejecutará sus programas.

Algunos de los lenguajes imperativos más difundidos en la actualidad son el *COBOL*, *BASIC*, *FORTRAN*, *Pascal*, *Modula 2* y *C*.

Una de las opciones para hacer más simple el trabajo de programar consiste en separar al programador del modelo computacional subyacente. De este modo se podrá concentrar en el desarrollo del programa como una *especificación* clara y concisa de la respuesta a un problema. Al elegir esta solución se está optando por un estilo *declarativo* de programación, llamado así porque lo que se hace es "declarar" (especificar) el problema que se quiere resolver en lugar de proporcionar la secuencia de acciones que el computador debe ejecutar para hallar la respuesta.

En lugar de indicar *cómo* operar, en programación declarativa se indica *qué* resolver. Por lo tanto, los lenguajes declarativos están más cerca del enunciado que de la resolución propiamente dicha. Por eso se les considera lenguajes de muy alto nivel.

Dentro del paradigma de *programación declarativa* se han desarrollado dos enfoques: la *programación funcional* y la *programación lógica*.

```
raíces(a,b,c)  =
    let discr ≡ sqr(b) - 4 * a * c
    in if discr ≥ 0
       then ((-b+sqrt(discr))/(2*a),(-b-sqrt(discr))/(2*a))
       else let (real, imag) ≡ (-b/(2*a), sqrt(-discr)/(2*a))
            in ((real,'+',imag,'i'),(real,'-',imag,'i'))
```

Figura 5.5.

Programación funcional

La programación en un lenguaje funcional consiste en construir funciones que el computador debe evaluar. La tarea primordial del programador es la construcción de una función que resuelva un problema dado; esta función puede depender de otras funciones subsidiarias y esta dependencia se expresa en una notación que obedece los principios matemáticos elementales. El papel del computador no es otro que el de evaluar las funciones e imprimir los resultados. Esta característica ha permitido que los lenguajes funcionales no dependan de ningún modelo particular de arquitectura, en contraste con la mayoría de los lenguajes imperativos.

La programación funcional tiene sus raíces en la lógica y en las matemáticas. En los programas funcionales se hace uso de conceptos y notaciones familiares a toda persona que tenga conocimientos elementales de matemáticas; de hecho, la derivación de programas funcionales a partir de su especificación se suele realizar mediante razonamiento ecuacional (aplicando las ecuaciones que definen las funciones implicadas en el programa).

Un programa funcional puede interpretarse, bien como un conjunto de definiciones (declaraciones) de propiedades o bien como un conjunto de reglas de computación. La primera interpretación la realiza el programador que está especificando (declarando) el problema, diciendo *qué* se busca en lugar de *cómo* se encuentra; por el contrario, el computador

entiende el programa como un conjunto de reglas de reescritura que le permitirán, finalmente, proporcionar la solución del problema.

Los lenguajes funcionales tienen muchas características atractivas, entre las que se encuentran:

- Los programas funcionales son bastante más cortos y legibles que los imperativos.
- Su fuerte base matemática hace más fácil razonar sobre la corrección de los programas.
- El significado de una expresión es su valor, que no cambia durante la ejecución del programa.

Uno de los primeros lenguajes funcionales que se desarrollaron fue el *LISP*, aunque no se considera completamente funcional por tener también características no funcionales. Otros ejemplos de lenguajes funcionales desarrollados en la actualidad son el *Hope*, *ML* y *Miranda*. El ejemplo de la figura 5.5 muestra, en el estilo de estos lenguajes, la solución de la ecuación de segundo grado.

Programación lógica

En programación imperativa, un programa no es otra cosa que el guión de las acciones que el computador debe seguir, y su ejecución consiste en obedecer esas órdenes. En *programación lógica*, un programa consiste en una serie de sentencias de la lógica matemática que describen un mundo, y su ejecución consiste en deducir algún objetivo propuesto.

Las sentencias mencionadas pueden ser de dos clases: *hechos*, similares a los axiomas de la lógica, o *reglas* de deducción, que definen relaciones entre objetos.

Las siguientes sentencias por ejemplo establecen una parte del *pedigree* de un perro:

hij(tobi, laika)
macho(tobi)
hembra(laika)
hij(tobi, pipo)
macho(pipo)
...

y podrían leerse como sigue: "Tobi es hijo/a de Laika", "Tobi es macho", etc. Establecen hechos ciertos sobre objetos concretos. Las reglas establecen también verdades, aunque se refieren a objetos genéricos,[1] por lo que tienen un carácter más general:

padre(P, H) <= hij(H, P) $\wedge$ macho(P)
madre(M, H) <= hij(H, P) $\wedge$ hembra(M)

Ahora, podría plantearse resolver el objetivo

hij(tobi, X)

que se pregunta por los X tales que "Tobi es hijo de X" (esto es, sus padres), y que se verifica para

X = laika
X = pipo

ofreciendo las dos soluciones. Ésta es una característica propia de los lenguajes lógicos: la posibilidad de dar muchas respuestas a un objetivo planteado.

También se puede plantear el objetivo

hij(X, pipo)

y se obtendría la relación de todos los hijos de este perro:

X = tobi
...

[1]Representamos los objetos concretos con palabras que empiezan en minúscula y los genéricos con palabras empezadas en mayúscula.

con lo que se observa otra peculiaridad de los lenguajes lógicos: los argumentos de una relación pueden representar datos o resultados de la misma, indistintamente. O ambas cosas: el objetivo

$$\text{hij(H, PM)}$$

ofrecería como respuestas los pares siguientes:

$$\begin{array}{lcl} \text{H = tobi} & ; & \text{PM = laika} \\ \text{H = tobi} & ; & \text{PM = pipo} \\ \ldots & ; & \ldots \end{array}$$

Otras reglas posibles son las siguientes:

$$\text{abuelo(A, N)} <= \text{padre(A, X)} \wedge \text{hij(N,X)}$$

que significa que "A es abuelo de N si A es padre de algún X y N es hijo de ese X". El par de reglas

$$\begin{array}{l} \text{antepasado(A, N)} <= \text{hij(N,A)} \\ \text{antepasado(A, N)} <= \text{hij(N,X)} \wedge \text{antepasado(A,X)} \end{array}$$

representa la disyunción siguiente: "A es un antepasado de N si N es su hijo (de A) o bien si N es hijo de algún X y A es un antepasado de ese X.

Este modelo de programación está fuertemente basado en la lógica de primer orden. El lenguaje de programación lógica por antonomasia es Prolog, del que se han desarrollado distintas versiones.

Programación orientada a los objetos

En la programación imperativa, un programa parte de unos cuantos datos iniciales, y evoluciona efectuando acciones que alteran los valores de esos datos. Como se ve, datos e instrucciones están separados de forma que, en principio, cualquier instrucción tiene acceso a cualquier objeto.

En cambio, en el enfoque de la programación orientada a los objetos (*POO*),[2] un programa es una descripción de los *objetos* que intervienen

[2]En inglés, OOP, Object Oriented Programming.

en él, como agentes interactuantes. Una descripción de objetos tiene dos aspectos: los valores de sus propiedades físicas, y las acciones (*métodos*) que describen su comportamiento, constituyendo una *clase* de objetos. Cada objeto es un miembro o instancia de su clase, y su actividad consiste en seguir los métodos definidos en la clase a la que pertenece. En el transcurso de su actividad, los objetos interactúan entre sí, pero el único modo de hacerlo es intercambiando *mensajes*.

Dado que cada objeto puede desenvolverse por separado, pueden evolucionar en paralelo, por lo que este enfoque es un modelo idóneo para la programación concurrente.

En el paradigma de la POO un programa se puede entender como un modelo *físico* que simula el comportamiento de una parte (real o imaginaria) del mundo. Esta perspectiva difiere de la imperativa, en la que se hace mayor hincapié en las manipulaciones de estructuras de datos o de modelos matemáticos.

La programación orientada a los objetos está más próxima a la Física que a las Matemáticas en el sentido de que, en lugar de describir una parte del mundo mediante ecuaciones matemáticas, lo que se hace es construir literalmente un modelo físico. Uno de los primeros lenguajes orientados a los objetos se llamó *ACTOR*, porque este paradigma nos presenta los objetos como actores sobre un escenario, dispuestos a interpretar su papel (sus *métodos*).

Algunas de las principales características de la programación orientada a objetos pueden resumirse así:

- *Encapsulación*

  Ya se ha dicho que las clases presentan dos aspectos: los valores que definen su estado, y las acciones, cuyo efecto puede recaer sobre el propio estado o emitir un mensaje a otros objetos.

  Estos dos aspectos se consideran locales, inaccesibles e inalterables por otros objetos y sólo se modifican por un efecto del comportamiento del propio objeto, o debido a la interacción con otro como consecuencia de algún mensaje recibido del mismo. Al unir datos

y acciones, la POO alcanza un nivel de abstracción superior al de la programación imperativa.

En un programa de simulación de una guerra naval, podría ser necesario definir la clase de las naves, caracterizadas en un momento dado por su posición (latitud y longitud), rumbo y velocidad que llevan, estado (desde intacto hasta destruido), etc. Son además capaces de navegar, método que altera su posición.

- *Herencia*

  A partir de un determinado objeto, se pueden crear otros que hereden sus datos y sus métodos, o incluso más especializados, contando con nuevos datos y métodos propios. Aparecen así verdaderas jerarquías de objetos, similares a las clasificaciones taxonómicas que se dan en las Ciencias Naturales.

  En el ejemplo anterior, los submarinos constituyen una especialización de las naves, incorporando como datos el número de torpedos que poseen y su estado (sumergidos o a flote), y como métodos sumergirse o salir a flote y disparar. Otra especialización posible es la clase de los buques, que son casos particulares de naves caracterizadas además por el número de misiles y de cargas de profundidad que llevan y capaces de disparar, aunque no pueden sumergirse.

- *Polimorfismo*

  Con frecuencia, objetos de clases distintas hacen uso de los mismos métodos. En la programación orientada a objetos es posible definir métodos llamados *polimórficos*, que pueden aplicarse sobre objetos de varias clases distintas. Las acciones concretas por ejecutar dependerán de la clase del objeto al que se apliquen.

  Con las clases naves definidas en los apartados anteriores, es posible definir métodos generales para dispararles, quizá deteriorando su estado hasta, incluso, la destrucción.

La metodología de resolución de problemas en programación orientada a objetos puede resumirse en los siguientes pasos:

1. Identificar los objetos que intervienen en el problema.
2. Caracterizar los dos aspectos de cada objeto: sus propiedades y mensajes asociados.
3. Establecer el estado inicial del escenario, así como la secuencia de mensajes que abre el telón y hace evolucionar la función teatral.

Algunos ejemplos de lenguajes de programación que soportan la programación orientada a los objetos son *Smalltalk* y *C++*. Otros lenguajes, están incorporando algunas de las características de la programación orientada a los objetos: por ejemplo, las últimas versiones de Turbo Pascal, de Borland ([Tur92]).

5.2 Descripción de los lenguajes de programación

Como dijimos en la introducción de este tema, los símbolos del vocabulario de un lenguaje se agrupan entre sí, formando estructuras sintácticas, que posteriormente son interpretadas por un traductor. En este apartado estudiamos cómo se describen los dos aspectos, sintaxis y semántica, de un lenguaje de programación.

Por ejemplo, diríamos que en castellano son válidas sintácticamente las "frases" compuestas por "sujeto" y "predicado"; en cuanto al significado, se entiende que el sujeto es quien realiza la acción, mientras que el predicado precisa la acción que se lleva a cabo.

En este caso especialísimo, la descripción de cierta estructura del castellano se ha llevado a cabo en el propio idioma castellano. Por el contrario, los lenguajes de programación no suelen describirse mediante lenguajes de programación, sino mediante los *metalenguajes*, llamados así porque son lenguajes cuyo cometido es describir otros lenguajes.

5.2.1 Sintaxis

La *sintaxis* estudia la forma de combinar los tokens de un lenguaje para formar frases correctas. Se concreta en una serie de *reglas* que

expresan las estructuras sintácticas válidas en el lenguaje. El reconocimiento de esas estructuras es necesario para dotarlas de significado y ejecutar las acciones que representan.

Ejemplos de esas reglas, dados en nuestro idioma y para él, son los siguientes:

- Un "sujeto" puede consistir en un "nombre propio" o un "artículo" seguido de un "nombre común".
- Los nombres propios son "Ana", "Ronda", ..., "Andalucía".

Para describir la sintaxis de un lenguaje se utiliza una secuencia de *símbolos terminales*, que son los elementos válidos de dicho lenguaje. Las aplicación de las reglas (también llamadas *producciones*), permiten sustituir un *símbolo no terminal* por una secuencia de símbolos, terminales o no. Un programa se deriva así partiendo de un único *símbolo inicial*, no terminal, que es el embrión del que surgen las frases por la aplicación sucesiva de las reglas sintácticas.

Siguiendo el ejemplo anterior, serían símbolos terminales las palabras "el", "la", ..., "Ana", etc. Algunos símbolos no terminales son "frase", "artículo" y "nombre propio". Entre ellos, "frase" es el símbolo inicial.

Esta sucinta explicación se comprenderá mejor examinando la notación BNF y los diagramas sintácticos, que introducimos seguidamente.

Notación BNF

Una de las formas más extendidas para describir la sintaxis de los lenguajes de programación es la notación *BNF* (Backus-Naur Form), utilizada por sus autores para el desarrollo del lenguaje Algol 60. Dado que la notación BNF es un lenguaje que nos permite describir la sintaxis de otro lenguaje, pertenece al grupo de los *metalenguajes*.

Para describir una producción en BNF hay que expresar la equivalencia de unos símbolos a otros. A tal fin se sitúa a la izquierda un símbolo no terminal y a la derecha la combinación de símbolos (terminales o no) equivalente, enlazados mediante otros símbolos (llamados a

veces metasímbolos), que no pertenecen al lenguaje sino a BNF. Dichos metasímbolos tienen el siguiente cometido:

- "<" y ">" delimitan los símbolos no terminales, y de esta forma poderlos distinguir de los terminales.
- El metasímbolo "|" se utiliza para separar varias combinaciones posibles, entre las que debe elegirse.
- El metasímbolo "::=" expresa la equivalencia dentro de las producciones, por lo tanto, separa el símbolo situado a su izquierda de una o más combinaciones de símbolos y metasímbolos situados a su derecha (separadas mediante "|"), y expresa que el primero puede sustituirse por la combinación de los segundos.

Por ejemplo, dadas las producciones:

<trapito> ::= <prenda> <color>
<prenda> ::= pantalón | camisa | sombrero
<color> ::= gris | marrón

<*trapito*> representa el símbolo inicial, <*prenda*> y <*color*> son los otros símbolos no terminales, y *pantalón*, *camisa*, *sombrero*, *gris* y *negro* son los símbolos terminales. Las frases válidas generadas por esa gramática son:

pantalón gris
pantalón marrón
camisa gris
camisa marrón
sombrero gris
sombrero marrón

Observaciones

- A veces resulta más cómodo distinguir los símbolos terminales y los no terminales escribiendo su inicial con minúscula y mayúscula, respectivamente, evitando el uso de "<" y ">". Por ejemplo, con la gramática

$$\begin{array}{lll} \text{Sílaba} & ::= & \text{Cons Voc} \\ \text{Cons} & ::= & \text{b} \mid \text{c} \mid \text{d} \mid ... \mid \text{z} \\ \text{Voc} & ::= & \text{a} \mid \text{e} \mid \text{i} \mid \text{o} \mid \text{u} \end{array}$$

resultan las siguientes sílabas válidas: "ba", "be", "bi", ..., "zi", "zo" y "zu".

- Como un símbolo no terminal cualquiera (digamos el L por ejemplo) puede aparecer en ambos lados de las producciones,

$$\begin{array}{lll} \text{I} & ::= & \text{aL} \\ \text{L} & ::= & \text{bL}|\text{c} \end{array}$$

sus sustituciones podrían generar estructuras que lo contengan:

$$\text{L} \rightarrow \ldots \text{L} \ldots$$

De este modo, una gramática puede resultar recursiva, generando un número infinito de frases:

"ac", "abc", "abbc", "abbbc", ...

Hoy día, todos los lenguajes de programación de alto nivel tienen gramáticas recursivas.

- Por otra parte, la notación BNF se puede ampliar (EBNF), usando los metasímbolos "[" y "]" para expresar una parte opcional y los metasímbolos "{" y "}" para abreviar la repetición de estructuras. Así por ejemplo, A[B]C expresa las frases "ABC" y "AC", y {A} representa bien la frase vacía o una de las siguientes: "A", "AA", "AAA", ...

Diagramas sintácticos

Otra forma, también muy extendida, para expresar las reglas sintácticas de un lenguaje es mediante *diagramas* o *grafos* sintácticos. La

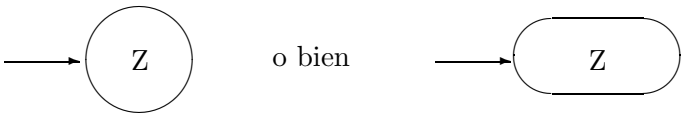
sintaxis del lenguaje Pascal, por ejemplo, fue inicialmente expresada de esta forma.

De acuerdo con esta representación, los símbolos terminales se representan dentro de círculos o rectángulos con las esquinas redondeadas, y el símbolo inicial y los símbolos no terminales dentro de cuadrados o rectángulos. Los sucesivos símbolos se unen entre sí mediante flechas, que indican el orden y sentido de producción.

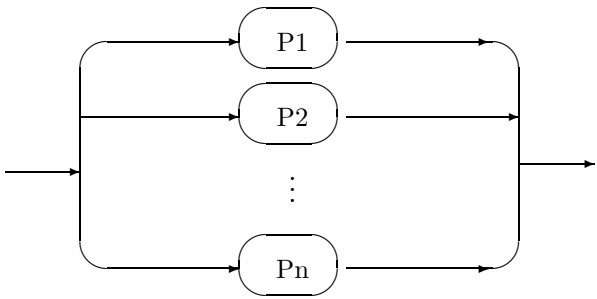
Veamos las principales representaciones:

- Símbolo inicial L

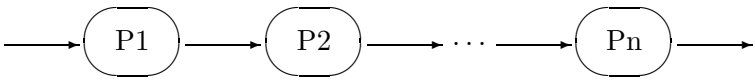
- Símbolo terminal Z



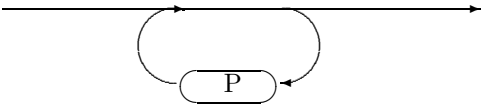
- $P1$ | $P2$ | ... | Pn



- $P1$ $P2$ ... Pn



- {P}



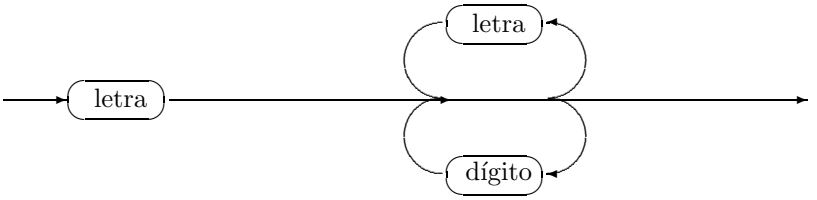
La utilización de los diagramas sintácticos facilita la construcción de frases correctas, pues es suficiente con seguir el sentido de las flechas e ir insertando los correspondientes símbolos según aparecen en el diagrama.

Recíprocamente, si al ir leyendo los símbolos de una frase se sigue uno de los caminos contemplados en el diagrama sintáctico, esa frase es correcta desde el punto de vista sintáctico.

Tanto la notación BNF como los diagramas sintácticos son equivalentes, si bien, en general éstos últimos son de más fácil comprensión y se aconsejan como punto de partida en la definición de lenguajes de programación.

Ejemplo Un identificador en Pascal se expresa mediante una letra aislada o seguida por una secuencia de letras o números o el carácter de subrayado, "_". Serían identificadores válidos los siguientes: "x", "y", "z", "altura", "longitud", "x1", "y1", "x_max", "y_max"

Su diagrama sintáctico sería el siguiente:



En BNF puede expresarse así:

Identificador	::=	Letra | Letra Letras_o_dígitos
Letras_o_dígitos	::=	Let_o_díg | Let_o_díg Letras_o_dígitos
Let_o_díg	::=	Letra | Dígito | Subr
Letra	::=	a | b | c | ... | z
Dígito	::=	0 | 1 | ... | 9
Subr	::=	_

5.2.2 Semántica

En el apartado anterior se han estudiado métodos para describir la sintaxis correcta de los lenguajes. La *semántica* de un lenguaje trata del *significado* asociado con los programas correctos en él, esto es, de su funcionamiento.

Es claro que lo más natural y rápido para explicar un lenguaje es proceder informalmente, a través de ejemplos o, más aún, matizando el exacto funcionamiento de cada construcción sintáctica (declaración, sentencia, etc.) No obstante, en los últimos años, ha surgido la necesidad de precisar la interpretación asociada con cada estructura válida (sintácticamente) en un lenguaje.

Se evitan así las ambigüedades inherentes a las descripciones informales de los lenguajes naturales y se consigue un entorno formal que facilita el estudio de propiedades de los programas, tales como la corrección o la eficiencia algorítmica.

Semántica operacional

En este enfoque, el significado de una estructura sintáctica viene dado por el modo con que la máquina lleva a cabo su evaluación o su ejecución. Así, la interpretación de un programa consiste en reproducir la historia de los cambios de estado que producen sus construcciones al interpretarse.

La semántica operacional de un lenguaje se define dando la función de transición ($\Rightarrow$) que asocia a una construcción sintáctica, en cada estado, el nuevo estado. Abreviadamente, escribimos

$$K : E \Rightarrow E'$$

para indicar que, al interpretar la construcción sintáctica K en el estado E, se origina el estado E'.

Por ejemplo, si representamos el estado inicial de una máquina mediante $\{\ldots, A = 2, B = 5, \ldots\}$, la ejecución de

$$
\begin{array}{l}
\textbf{begin} \\
\quad A := A + B; \\
\quad B := A - B; \\
\quad A := A - B \\
\textbf{end}
\end{array}
$$

produce la siguiente computación:

$$
\begin{array}{rcl}
 & \left\{ \begin{array}{l} \textbf{begin} \\ \quad A := A + B; \\ \quad B := A - B; \\ \quad A := A - B \\ \textbf{end} \end{array} \right\} & : \left\{ \begin{array}{l} \ldots \\ A = 2 \\ B = 5 \\ \ldots \end{array} \right\} \\
\Rightarrow & \left\{ \begin{array}{l} \textbf{begin} \\ \quad B := A - B; \\ \quad A := A - B \\ \textbf{end} \end{array} \right\} & : \left\{ \begin{array}{l} \ldots \\ A = 7 \\ B = 5 \\ \ldots \end{array} \right\} \\
\Rightarrow & \left\{ \quad A := A - B \quad \right\} & : \left\{ \begin{array}{l} \ldots \\ A = 7 \\ B = 2 \\ \ldots \end{array} \right\} \\
\Rightarrow & & : \left\{ \begin{array}{l} \ldots \\ A = 5 \\ B = 2 \\ \ldots \end{array} \right\}
\end{array}
$$

La semántica operacional puede centrarse en los detalles más finos de la ejecución (semántica operacional *estructural*) o ignorarlos (semántica

natural):

$$\left\{\begin{array}{l}\textbf{begin}\\ \quad A := A + B;\\ \quad B := A - B;\\ \quad A := A - B\\ \textbf{end}\end{array}\right\} : \left\{\begin{array}{l}\cdots\\ A = 2\\ B = 5\\ \cdots\end{array}\right\} \Rightarrow \left\{\begin{array}{l}\cdots\\ A = 5\\ B = 2\\ \cdots\end{array}\right\}$$

Semántica denotacional

El significado de una construcción sintáctica viene dado por el efecto que produce. En lugar de explicar *cómo* se efectúa cada construcción sintáctica, sólo interesa ahora *qué* significa. En términos lingüísticos, puede considerarse la función semántica como una función que asocia un significado a cada significante.

Los significantes son las posibles construcciones válidas del lenguaje, y están definidos por las reglas sintácticas, agrupados en distintas categorías: declaraciones, expresiones, instrucciones, etc. Los significantes son objetos matemáticos, agrupados en distintas categorías, correspondientes a las categorías sintácticas.

Por ejemplo, las *cantidades* numéricas serán el significado asociado a literales, variables, expresiones y funciones numéricas; puesto que el efecto de una instrucción de asignación consiste en alterar el estado de la memoria, su semántica es una función entre estados de la memoria, etc.

Más en general, una instrucción puede alterar la memoria, la entrada y la salida, que podrían consistir matemáticamente en un vector y sendas listas. Si representamos esas tres entidades mediante M, I y O, el *estado* de la máquina será la terna $E = M \times I \times O$.

Entonces, podemos precisar el significado de una instrucción así:

$$S[\![Instr]\!] : E \to E.$$

Concretamente, tenemos:

- $S[\![x \leftarrow x_0]\!](M, I, O) = (M', I, O)$

donde M' es el estado de la memoria M, sustituyendo el valor asociado con el identificador x por x_0.

- $S[\![write(x_0)]\!](M, I, O) = (M, I, O')$
  donde O' es el estado de la salida O, añadiendo el valor x_0.
- $S[\![read(x)]\!](M, I, O) = (M', I', O)$
  donde M' se obtiene al sustituir x por x_0 en M (siendo x_0 el primer dato de la entrada), y además $I = (x_0, x_1, \ldots)$ y $I' = (x_1, \ldots)$.

Como se ve, el efecto asociado a cada construcción K se precisa mediante una función, $S[\![K]\!]$, llamada su *función semántica*.

Siguiendo con el ejemplo anterior, podemos interpretar la instrucción compuesta mediante la composición de funciones, así:

$$S[\![\textbf{begin } A := A + B; B := A - B; A := A - B \textbf{ end}]\!]$$
$$= S[\![A := A + B]\!] \circ S[\![B := A - B]\!] \circ S[\![A := A - B]\!].$$

En el caso de las instrucciones, el *dominio* y *codominio* de esas funciones semánticas se representan mediante el estado de la máquina; para interpretar las demás construcciones sintácticas (tales como las declaraciones, los identificadores, los literales o las expresiones) se necesita igualmente formalizar sus conjuntos dominio y codominio asociados. Los modelos matemáticos correspondientes a estas entidades se llaman *denotaciones*, de donde procede el nombre de este enfoque de la semántica.

Semántica axiomática

Este enfoque atiende a las "propiedades" que tiene la interpretación de cada sentencia I. Estas propiedades se expresan a través de *aserciones* sobre el estado de la máquina abstraída, antes y después de esa sentencia. Se escribe

$$\{C_0\} \quad I \quad \{C_1\}$$

para expresar que, si se verifica C_0 al comienzo de la instrucción I, cuando ésta termine[3] se verificará C_1. La condición (C_0) que describe el

[3]Obsérvese que no se asegura que en las condiciones C_0 la instrucción I termine, sino sólo que llegaremos a C_1 "si ésta termina"

estado previo a I se llama *precondición*, y la *postcondición* (C_1) se refiere al estado tras I.

En el ejemplo anterior, podemos escribir lo siguiente:

$$
\begin{array}{l}
\{A = A_0 \wedge B = B_0\} \\
\textbf{begin} \\
\quad A := A + B; \\
\quad B := A - B; \\
\quad A := A - B \\
\textbf{end} \\
\{A = B_0 \wedge B = A_0\}
\end{array}
$$

Por lo tanto, la semántica axiomática nos permite estudiar las propiedades que verifica un programa, gracias al empleo de la lógica de predicados.

5.3 Procesadores de lenguajes

Aunque, hablando estrictamente, el único lenguaje que los computadores comprenden es el código máquina, pueden ejecutar también programas escritos en lenguajes de alto nivel, gracias a los *traductores*, programas que convierten programas escritos en lenguajes evolucionados en programas en código máquina:

$$
\begin{array}{ccccc}
 & & \text{programa} & & \text{programa} \\
\text{traductores} & : & \text{en un lenguaje} & \rightarrow & \text{en lenguaje} \\
 & & \text{evolucionado} & & \text{de máquina}
\end{array}
$$

En general, un traductor es un programa que convierte un programa, escrito en un lenguaje, en un programa equivalente, escrito en otro lenguaje. Llamamos *programa fuente* y *programa objeto* respectivamente al programa inicial y al final. Estos términos se aplican también a los lenguajes respectivos.

El proceso de traducción se lleva a cabo en varias fases; en cada una de ellas, pueden aflorar algunos de los errores, de diversa índole, cometidos por el programador. Por otra parte, una componente esencial

que manejan estas etapas es la *tabla de símbolos*, estructura de datos que contiene los identificadores que intervienen en el programa, junto con su descripción. Las etapas mencionadas son básicamente las siguientes:

- *Análisis léxico.*- En esta fase, el texto del programa se traduce en una lista de símbolos, llamados *tokens*, tales como: el símbolo de abrir paréntesis, el símbolo de asignación, el identificador 'pi', el número entero -71, etc.

  La escritura incorrecta de identificadores (por ejemplo, empezando por un dígito en lugar de un carácter) es uno de los errores léxicos más frecuentes.

- *Análisis sintáctico.*- Su misión consiste en recorrer la lista de tokens, que le va proporcionando el analizador léxico, y tratar de agruparlos para reconocer "frases" o estructuras sintácticas, tales como declaraciones, sentencias o expresiones, siguiendo las reglas del lenguaje (véase 5.2.1). Por ejemplo en la figura 5.6:

  El uso no equilibrado de paréntesis o de otros delimitadores son errores sintácticos frecuentes.

- *Acciones semánticas.*- Un programa no es sólo un reconocedor de estructuras gramaticales: debe llevar a cabo las acciones indicadas por el programa fuente. Estas acciones representan el significado del programa, y son tales como las siguientes:

  - Dar de alta un identificador en la tabla de símbolos y reservar espacio de memoria para su descripción.
  - Consultar la tabla de símbolos y extraer la descripción asociada a un objeto.
  - Comprobar la compatibilidad de tipo entre los objetos que aparecen en el programa.

  Ejemplos de errores asociados con las acciones mencionadas son los siguientes:

  - Intentar definir un identificador ya definido.

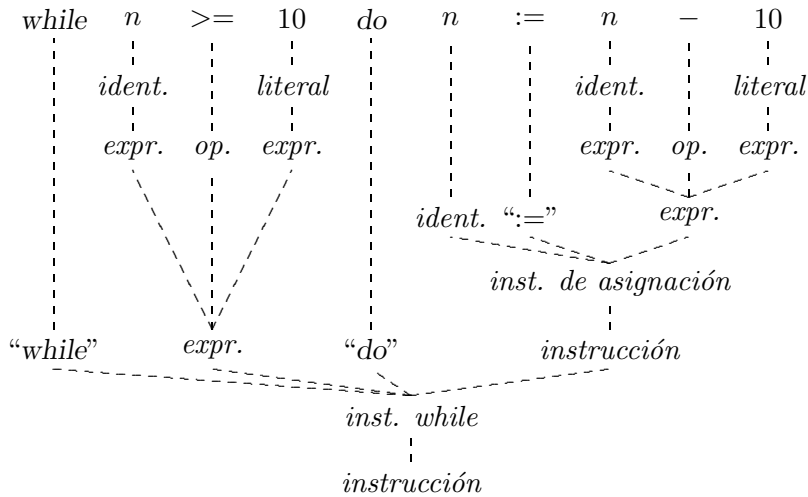


Figura 5.6. Reconocimiento sintáctico de una sentencia

- Tratar de usar un identificador desconocido.
- Escribir una operación imposible entre sus operandos, debido a su tipo.

Entre las acciones mencionadas, el aspecto de la verificación de tipos está muy desarrollado en los lenguajes evolucionados, ya que así se evita un gran número de errores durante el desarrollo de los programas. Los lenguajes que llevan a cabo este tipo de comprobaciones se dicen *fuertemente tipados*; se dice que la verificación de tipos es *estática* cuando se lleva a cabo antes de la evaluación.

- *Generación de código.*- Esta fase consiste en producir una representación del programa coherente con la sintaxis y semántica dadas en las etapas anteriores. Esta representación puede consistir directamente en el código máquina, aunque por lo general se obtendrá previamente una versión en lenguaje ensamblador e incluso a veces, antes aún, una representación del programa (llamada *código*

intermedio) apropiada para facilitar la traducción final.

- *Optimización.*- En esta etapa se efectúan manipulaciones sobre el código intermedio o sobre el código objeto final, con el objeto de mejorar la calidad del código generado. Por lo general, estas mejoras consisten en disminuir el tamaño del código generado y en hacerlo más eficiente, sin alterar a cambio el efecto que deba producir el programa.

5.3.1 Compiladores e intérpretes

Existen dos modos de traducción, comparables en todo a las dos modalidades de traducción siguientes, entre las diferentes lenguas humanas:

- En el proceso de *compilación*, una editorial parte del volumen original, escrito en el idioma *fuente*, y produce el volumen correspondiente, escrito en el lenguaje *objeto*. Los lectores en el idioma objeto interpretan el texto sólo después de estar completamente traducido.

- La *interpretación* es la traducción simultánea, por ejemplo en una convención internacional: un intérprete traduce fragmentos pequeños de frases en el idioma fuente al idioma objeto, a medida que las escucha. Los congresistas reciben e interpretan la traducción a medida que ésta se produce.

Entre ambas modalidades, existen traductores mixtos, que en una primera fase desarrollan una semicompilación, que incluye diversas comprobaciones léxicas y sintácticas, y produce una representación del programa fuente en un código intermedio; y en una fase posterior ejecutan la interpretación de ese código intermedio.

5.3.2 Entornos de programación

Para crear un programa ejecutable en el computador hay que realizar una serie de etapas que, tradicionalmente, se venían efectuando con programas independientes. Entre ellos se pueden citar los siguientes:

- Editores

  Son programas que permiten el trabajo con textos, que sirven para crear los programas fuente. Debido a que, por lo general, trabajan en codificación ASCII pura, carecen de algunas de las facilidades de los procesadores de textos, tales como el uso de diferentes tipos de letra, o la gestión de títulos, subtítulos y párrafos.

- Compiladores

  Los compiladores, como se ha explicado en este mismo tema, son los programas que realizan la traducción del *programa fuente* generando el *programa objeto*. Tradicionalmente, era frecuente que el proceso de compilación se realizase en varias fases, y que el compilador estuviera constituido por varios módulos, de forma que, si no se presentan errores, las distintas fases de la compilación se sucedan secuencialmente, al ser ejecutados los distintos módulos del compilador mediante un fichero de proceso por lotes.

- Enlazadores

  Los programas fuente suelen contener llamadas a funciones predefinidas del lenguaje. En general, estas funciones se encuentran precompiladas en unas *bibliotecas* que se suministran junto con el traductor del lenguaje, que se pueden adquirir o desarrollar por separado. A veces el propio programa fuente se va compilando por partes, módulos, o unidades. El programa enlazador (*linker*) se encarga de incorporar al programa objeto el código correspondiente a las llamadas a funciones y de unir los distintos módulos precompilados.

- Depuradores

  Algunos de los errores que pueden aparecer en programación son muy difíciles de detectar. La ejecución del programa puede no proporcionarnos la información suficiente para saber con seguridad donde está el error. Los programas de depuración (*debuggers*) permiten hacer un seguimiento preciso de la ejecución de las instrucciones, y de esta forma aislar y detectar los errores. Para ello, los

programas de depuración toman el control de la ejecución del programa y, entre otras posibilidades, permiten: ejecutar el programa paso a paso; establecer puntos de detención del programa; generar una relación de las instrucciones ejecutadas en un cómputo determinado (*traza* del programa); conocer en cada momento los valores de las variables que nos interesen (*tablas de seguimiento*), etc.

En la actualidad, es frecuente que algunos o todos estos programas que solían ser independientes se agrupen en un único programa que pasa a constituir un verdadero entorno de programación.

Si la integración entre los distintos componentes del entorno es alta, se acelera notablemente el proceso de creación de programas, y se puede disponer de recursos y ayudas a la programación que serían impensables en programas separados. Entre ellos podemos destacar:

- Ayudas a la edición

  Los entornos integrados presentan interesantes ayudas a la edición permitiendo, por ejemplo, abrir varios programas a la vez y copiar bloques de uno a otro, buscar y sustituir grupos de caracteres, etc.

- Sensibilidad al contexto

  Esta es una característica de muchos editores de los entornos integrados, por la que, durante la escritura del programa el editor detecta los distintos tipos de tokens del lenguaje (tales como símbolos y palabras reservadas, constantes, etc.), resaltándolas mediante el color, el tipo de letra o la intensidad, tanto en pantalla como en las copias impresas. Este es el caso del compilador de *Turbo Pascal*, a partir de la versión 7.0.

  En algunos entornos integrados sencillos se realiza el análisis sintáctico al terminar cada construcción sintáctica, rechazándose las incorrectas. Un ejemplo de la ayuda proporcionada consiste en el equilibrado de ciertos delimitadores, tales como los paréntesis, emparejándose apropiadamente o detectándose el error.

- Ayuda contextual

Los entornos integrados disponen de un sistema de ayuda interactiva, con detección del contexto, tanto para los mandatos del entorno como para las instrucciones del lenguaje. Al realizar una petición de ayuda aparece en pantalla una explicación precisamente del mandato que estamos utilizando.

- Compilación en la memoria principal

  Para conseguir una mayor velocidad durante el proceso de depuración y puesta a punto de un programa, los entornos integrados tienen la posibilidad de compilar los programas en la memoria principal. Una vez que el programa está terminado, puede compilarse en disco, con el simple cambio de una opción. Las distintas fases de compilación, enlazado y ejecución se suceden sin que el usuario lo advierta, obteniéndose un alto nivel de interacción y de productividad, al reducirse los tiempos de espera.

- Depuración integrada

  Los entornos de programación incluyen, por lo general, características propias de los programas de depuración avanzados como las que hemos mencionado anteriormente. La gran ventaja es que dichas opciones están disponibles dentro del propio entorno, por lo que no es necesario cambiar de programa.

- Gestión de proyectos

  Los entornos de programación permiten la gestión de grandes programas, mediante la *compilación separada* por *módulos* o *unidades*. Al compilar el programa principal, el gestor de programas, ordena la recompilación solamente de los módulos que han sido modificados desde la última compilación. Esta compilación separada facilita la utilización de bibliotecas especializadas (para gráficos, cálculo numérico, etc.) y también el desarrollo de programas por distintos grupos de trabajo.

5.4 Ejercicios

1. Siguiendo el ejemplo de las figuras 5.3 y 5.5, describa en ambos estilos el cálculo del área de un triángulo, conociendo las medidas de los lados, aplicando la fórmula de Herón:

$$A = \sqrt{p(p-a)(p-b)(p-c)}$$

siendo p el semiperímetro del triángulo:

$$p = \frac{a+b+c}{2}.$$

2. Siguiendo el ejemplo de la genealogía perruna, defina la relación "hermanos", que se da entre dos perros distintos con los mismos padres.

3. Para escribir una lista de nombres en castellano, los sucesivos elementos de la lista se separan entre sí con comas y el último con la conjunción "y". También son válidas las listas con un único nombre, que naturalmente se escribe sin separador alguno.

   Por ejemplo, si consideramos los nombres Ana, Juan y Elisa, resultan válidas las siguientes listas:

   Ana
   Juan y Elisa
   Elisa, Juan y Ana
   Ana, Ana, Ana, Ana y Ana

   Exprese esa sintaxis mediante BNF y diagramas sintácticos.

4. Para escribir un número en cierta calculadora, se aplican las siguientes reglas:

   (a) Un número se expresa mediante una cantidad, que puede ser entera o decimal, y que puede estar precedida o no por un signo "+" o "-".

   (b) Una cantidad entera se expresa mediante uno o más dígitos

   (c) Una cantidad decimal se puede expresar con uno o más dígitos (parte entera), a continuación una coma y después uno o más dígitos (parte decimal). También es posible expresar cantidades decimales sin parte entera o sin parte decimal, aunque debe aparecer al menos una de ellas, además de la coma.

Por ejemplo, son válidos los siguientes números:

-1	,7
+6,	,368
+,8852	-123,456

(a) Exprésela mediante diagramas de flujo

(b) Exprésela mediante EBNF

(c) Genere unas cuantas sentencias válidas

5. En el ejemplo de semántica axiomática sobre el intercambio de valores entre variables numéricas, precise las aserciones intermedias C_1 y C_2:

begin
$\{A = A_0 \wedge B = B_0\}$
$A := A + B;$
$\{...C_1...\}$
$B := A - B;$
$\{...C_2...\}$
$A := A - B$
$\{A = B_0 \wedge B = A_0\}$
end

6. Estudie las facilidades que ofrece la última versión del entorno de programación de Turbo Pascal ([Tur92]). Concretamente, relacione las que posee para la depuración de programas.

5.5 Comentarios bibliográficos

El área de los lenguajes de computador es realmente amplia y profunda. Es inútil pues pretender reseñar siquiera una pequeña parte de las referencias más significativas sobre aspectos tan diversos como el estudio de las gramáticas y la teoría de autómatas, la semántica, el desarrollo de traductores y entornos de programación y los diferentes paradigmas. Además de inútil, sería vano pretenderlo en un texto introductorio y general como es éste, incluso limitándonos al terreno de las aplicaciones por profesionales no informáticos.

Encontramos que el estudio de los distintos mecanismos expresivos comunes a muchos lenguajes es preferible desde la práctica de algún lenguaje concreto (como Pascal, por ejemplo) o desde el desarrollo de algoritmos con un lenguaje

de especificaciones. En el segundo tomo de este texto adoptamos el primer punto de vista; en [GGSV93] se adopta el segundo.

Para ampliar conocimientos sobre los entornos de programación, tampoco parece haber nada tan recomendable como empezar por uno concreto. Las últimas versiones de *Turbo Pascal*, de Borland ([Tur92]) son ejemplares en ambos aspectos, lo que explica su gran difusión, tanto en el ambiente educativo como en el de las aplicaciones.

Naturalmente, la formación lograda así dotará al estudiante de un punto de vista imperativo. No es éste un grave inconveniente (ya que éste ha sido y sigue siendo todavía el paradigma de mayor difusión) siempre que su educación haya transcurrido por la senda de la disciplina y los buenos hábitos y métodos.

Sin embargo, no son pocas las aplicaciones actuales que adoptan otro enfoque distinto del imperativo. Aunque cada modelo introducido es de por sí un mundo, damos algunas referencias de ellos. En [Tes84] se ofrece una vista panorámica sobre los lenguajes de programación de la pasada década. Un libro clásico sobre programación en lenguaje Pascal es [Gro86]. Sobre programación funcional, [BW88] y [Bai90] son buenas introducciones, respectivamente en los lenguajes *Miranda* y *Hope*. En [GGSV93] se consideran ambos enfoques, el imperativo y el funcional. Una buena introducción al Prolog puede hallarse en [CM87]. Para adentrarse en este bello lenguaje, debe leerse [SS86]. La filosofía orientada a los objetos se extiende con rapidez, por lo que también las referencias sobre el tema proliferan. [Pas86], [Tho89] y [Weg89] son artículos introductorios a este modelo de programación.

Finalmente, en [Wir86a] puede verse una explicación ampliada de la notación BNF y diagramas de flujo, así como una introducción a los problemas que plantea el análisis sintáctico y la compilación de programas en lenguajes de alto nivel.

Capítulo 6

Bases de datos

6.1 Bases de datos y SGBD

6.1.1 Archivos y SGA

En el capítulo 2 se vio la posibilidad que tienen los computadores de trabajar con datos elementales (tales como números reales, enteros, caracteres y valores lógicos) o de organizarlos en estructuras más complejas, tales como vectores o tablas. En el 4 se introdujeron los archivos como organización básica de la información, almacenada en un dispositivo físico; entonces no fue preciso distinguir en ellos ningún tipo de organización: podían considerarse como listas de elementos de caracteres.

Ahora interesa considerar un *archivo* como una lista de datos del mismo tipo, cada dato de un archivo es un *registro*, y puede tener una estructura compuesta por varios datos de distinto tipo, llamados *campos*. Considerando el fichero de los alumnos de un colegio como símil

de archivo, cada ficha es un registro, y cada dato consignado en ella constituye un campo.

La utilización de los archivos en informática ha permitido desarrollar programas de aplicación en áreas muy diversas: el fichero de socios de una biblioteca, el de su fondo bibliográfico, el de clientes de cualquier empresa, el de proveedores, el de artículos de un inventario, etc. Los programas desarrollados para gestionar archivos se llaman en general *sistemas de gestión de archivos* (*SGA*). Debido a su directa aplicabilidad, la utilización de los archivos y los SGA han evolucionado muchísimo en los últimos años, originando las bases de datos y, correspondientemente, los programas dedicados a su gestión.

6.1.2 Bases de datos y SGBD

Hablando en términos muy amplios, una *base de datos* consiste en una colección de datos, organizados de forma integrada en archivos, junto con un conjunto de programas dedicados a su gestión. Estos programas forman los *sistemas de gestión de bases de datos* (*SGBD*).

Al igual que los SGA, los SGBD son programas destinados principalmente a almacenar, manipular y recuperar la información, y desempeñan operaciones que suelen tener lugar a diario: altas y bajas de clientes, apuntes en las cuentas bancarias, adquisición y pérdida de libros o su préstamo y devolución, elaboración de informes sobre los clientes o socios morosos, sobre las calificaciones de los alumnos de un curso en una asignatura y un largo etcétera.

Sin embargo, en los SGA el programador debía controlar artesanalmente múltiples detalles, tales como la descripción pormenorizada de los datos, su almacenamiento en los dispositivos, la concordancia de cada rutina que maneje esos datos con la descripción de los mismos, la concordancia de los datos cuando se repitan en distintos archivos, etc. Más aún, el programador es responsable de que cada modificación de los datos o de los programas se vea reflejada en todos esos puntos.

En cambio, los SGBD controlan automáticamente operaciones como el almacenamiento de los datos en los dispositivos físicos, así como el

acceso de los programas a los mismos; además, aseguran la consistencia de los datos repetidos y de los programas que los manejan, etc.

Gracias a estas ventajas, con los SGBD es posible concentrar más atención en el problema (a su nivel) y mucha menos en los detalles (de bajo nivel) de su implantación física (o interna), lo que facilita enormemente el diseño, desarrollo y mantenimiento de las bases de datos y permite, por lo tanto, afrontar otras de gran envergadura.

Conclusión

En resumen, los SGA consisten en agregados de programas y archivos, añadidos por el programador para cubrir ad hoc necesidades nuevas. En cambio, los SGBD son sistemas "integrados" de programas y archivos, diseñados con el propósito específico de desarrollar y gestionar bases de datos, como indica su nombre, de acuerdo con un plan general.

Por ello, la funcionalidad de los SGBD es superior a la de los SGA. Más concretamente, se pueden identificar las siguientes diferencias entre ambos tipos de sistemas:

- En general, los SGA consisten en aglomerados de programas y archivos rígidos, puesto que se han agregado sucesivamente para resolver sólo operaciones muy concretas. Por el contrario, los SGBD se caracterizan por una gran *flexibilidad*, permitiendo la reestructuración de la información gestionada, la incorporación de nuevas aplicaciones, o la modificación de las existentes manteniendo la organización global.

- La información de los SGA presenta frecuentemente *redundancias* innecesarias. La repetición de los datos en los diversos archivos es *ineficiente*: además de hacer un mal aprovechamiento de la memoria, aumenta y dificulta a la vez las operaciones de consulta y mantenimiento de esa información. La redundancia de los datos lleva aparejado además el riesgo de que esos datos repetidos no concuerden (*inconsistencia*), lo que se supera en los SGBD gracias a su alto grado de integración.

- Por lo común, los SGA están implantados en sistemas operativos monousuario, por lo que carecen de mecanismos para atender a diferentes personas desde diferentes puntos. En cambio, los SGBD están diseñados muchas veces para funcionar en sistemas multiusuario, pudiendo atender consultas y actualizaciones de los datos que se solicitan al mismo tiempo, y están dotados con mecanismos para garantizar la *seguridad* y *confidencialidad* de los datos en sistemas *multiusuario*.

- Los SGA carecen en general de *protección frente a fallos* del sistema. En cambio, los SGBD disponen de los mecanismos necesarios para recuperar la información en caso de necesidad.

Por consiguiente, los SGBD aventajan a los SGA en muchos aspectos. Aunque se ha señalado el inconveniente de que, al ser sistemas más complejos y potentes, tienen mayores requerimientos de hardware y software, este problema se está atenuando con el aumento de potencia que los computadores personales están experimentando hoy en día y con el abaratamiento del software.

A pesar de las ventajas que suponen las bases de datos, debe advertirse que el buen diseño de una base de datos no es una cualidad intrínseca de los SGBD, así como tampoco son propios de los SGA los defectos de un mal diseño o desarrollo. Una base de datos desarrollada con un SGA puede resultar segura, funcionar en un entorno multiusuario y estar exenta de redundancias (en la información) aunque, eso sí, debe ser el desarrollador quien se preocupe de mantener esas cualidades, mientras que un SGBD asume algunas de esas responsabilidades y facilita muchas otras.

6.1.3 Niveles de una base de datos

La complejidad de una base de datos puede superarse dotando al sistema de una organización lógica que facilite su manejo. Según las normas del ANSI/SPARC[1] ([Stu75]), esta organización debe constar de

[1] American National Standard Institute.

los tres niveles siguientes, de más próximo al problema a más cercano a la máquina:

- Nivel *interno* o *físico*, formado por los detalles de la organización física de la base de datos, tales como la representación escogida de los datos a bajo nivel y el almacenamiento real de los datos en los discos u otros dispositivos físicos.

- Nivel *conceptual* o *lógico* donde se describe, a alto nivel, la información de la base de datos, ignorando los detalles de su almacenamiento.

- *Nivel externo* o *de visión*, que comprende las visiones particulares que tienen de la base de datos los distintos usuarios, ya sean usuarios finales o programadores, con frecuencia en distintos lenguajes. Estos últimos ven la descripción de la información en sus lenguajes respectivos.

6.2 El modelo entidad-relación

Un modelo de datos es un enfoque adoptado para formalizar la información del mundo real que atañe a una base de datos. En el nivel conceptual y lógico, existen dos grupos de modelos: los basados en objetos, y los basados en registros.

El modelo *entidad-relación* es seguramente el más ampliamente aceptado entre los primeros. En él se concibe la realidad como un conjunto de objetos (llamados *entidades* y caracterizadas por *atributos*) y *relaciones* entre los mismos. En los ejemplos mencionados, constituye una entidad cada cliente concreto, cada cuenta bancaria, cada socio de la biblioteca, cada uno de sus libros, etc. Resulta natural agrupar las entidades en grupos homogéneos (*conjuntos de entidades*): en el conjunto de entidades "libros", por ejemplo, tendrían cabida todos los libros de la biblioteca. Considerando ahora este conjunto de entidades, podemos identificar en ellas los atributos siguientes: título, autor, editorial, signatura, número de registro, etc.

Las asociaciones entre varios conjuntos de entidades pueden establecerse mediante relaciones. Por ejemplo, se puede considerar una relación entre los conjuntos de entidades "socios"-"libros" para expresar los "préstamos". También es posible caracterizar las relaciones con atributos (como la "fecha" de devolución para un préstamo).

Las relaciones entre entidades pueden ser de varios tipos:

- Una relación entre dos entidades es de *uno a uno* (abreviadamente, 1:1) si existe una correspondencia biunívoca entre ellas. Por ejemplo, cada automóvil tiene una única matrícula y viceversa.

- La relación "socios"-"libros" mencionada en el párrafo anterior es de *uno a muchos* (1:m): un socio puede tener prestados desde ninguno hasta muchos libros al mismo tiempo, pero cada libro tiene, a lo sumo, un único socio que lo ha pedido.

- La relación entre "profesores"-"asignaturas" es de *muchos a muchos* (m:m), ya que un profesor puede impartir varias asignaturas, e igualmente una asignatura puede ser impartida por varios profesores. En este tipo de relaciones no se imponen restricciones sobre el número de elementos por relacionar de cada conjunto de entidades.

Con estos conceptos, es posible representar gráficamente la estructura de una base de datos del siguiente modo:

- Cada conjunto de entidades se expresa mediante un rectángulo.

- Los atributos que caracterizan un conjunto de entidades se representan con elipses asociadas al rectángulo correspondiente.

- Las relaciones se representan mediante rombos, unidos con líneas a los conjuntos de entidades asociados. En los casos de las relaciones 1:1 y 1:m, el conjunto de entidades dado por el 1 se señala desde la relación con una flecha:

  – *Relación m:m*

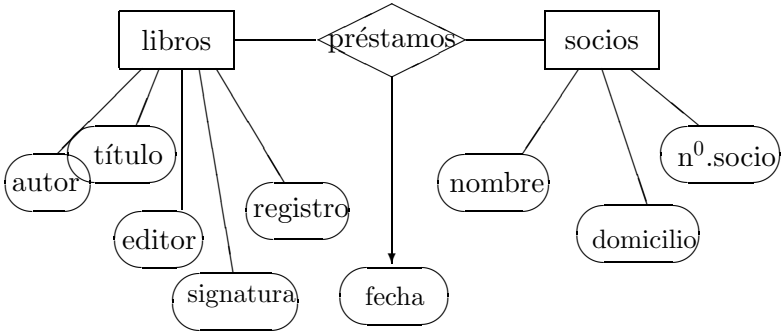
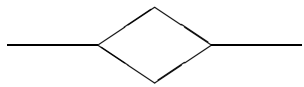
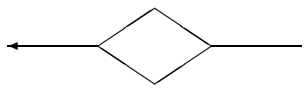


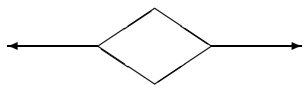
Figura 6.1. Representación gráfica de una relación



– *Relación 1:m*



– *Relación 1:1*



Por ejemplo, la relación "préstamos" puede representarse como muestra la figura 6.1.

6.3 Modelos de bases de datos basados en registros

Tradicionalmente, se han considerado diferentes enfoques a la hora de organizar una base de datos. Cada uno adoptaba una concepción distinta de la realidad, dependiendo del tipo de estructura de datos más adecuado para construir un modelo del problema planteado.

En el modelo *jerárquico* subyace una estructura de datos arborescente. Ejemplos de esta estructura son la genealogía de los toros de lidia y la organización de directorios de un disco. Esta orientación se suele usar cuando el problema abarca sólo relaciones 1:m.

El enfoque *en red* adopta como modelo de datos el concepto matemático de *grafo*: un conjunto de puntos llamados *nodos*, que se conectan unos con otros mediante líneas llamadas *arcos*. El ejemplo arquetipo de grafo es una red de carreteras: los arcos son los diferentes tramos, y los nodos las intersecciones. En este tipo de relaciones se pueden representar también las relaciones m:m, aunque no de un modo muy natural.

Al igual que en el modelo entidad-relación, en el enfoque de datos en red la base de datos consiste en un conjunto de *entidades* (que son los nodos de un grafo), caracterizadas por *atributos* y relacionadas mediante *relaciones* (que son los arcos). En el modelo en red, a las relaciones se les llama *ligaduras*, y son siempre binarias, si bien es posible construir relaciones de cualquier orden combinando dichas relaciones binarias.

En la actualidad, los modelos jerárquico y en red se consideran ampliamente superados por el modelo *relacional*, que está muy desarrollado y extensamente difundido, funcionando en equipos grandes y pequeños. Debido a su importancia, le dedicamos el siguiente apartado.

6.3.1 El modelo relacional

El enfoque relacional considera una base de datos como un conjunto de *tablas*,[2] que representan entidades o relaciones. Dados n conjuntos de

[2]Llamadas *relaciones*, de donde viene el nombre de este modelo.

entidades $E_1, ..., E_n$, una tabla es un conjunto de n-tuplas de la forma $(e_1, ..., e_n)$, donde $e_i \in E_i$; en otras palabras, una relación es un subconjunto del producto cartesiano $E_1 \times ... \times E_n$.

Visto un conjunto de entidades como una tabla, cada fila representa una entidad concreta, y cada columna un atributo:

libros

Autor	Título	Editor	Signatura	Registro
...	...	...	...	...

Si la tabla representa una relación, cada columna es una de las entidades vinculadas por ella. En realidad, no es preciso consignar cada entidad completa, sino que basta con incluir los atributos necesarios para determinarla unívocamente. Este conjunto de atributos se llama *clave* de la entidad:

préstamos

Registro	Núm. socio	Fecha
...	...	...

Pongamos un ejemplo. Una base de datos correspondiente a la información de cierta carrera universitaria[3] podría constar, entre otras, de dos relaciones, que almacenan respectivamente los datos personales de los alumnos y las asignaturas en que se han matriculado en un curso:

Direcciones (nombreAl, dirección, teléfono)

Matrículas (nombreAs, nombreAl, curso, númCréditos, nota)

El atributo *nombreAl* es una clave para la primera relación. Una clave de la segunda la forman los atributos *nombreAs* y *nombreAl*.

En realidad, no se han establecido las relaciones concretas entre las entidades (para lo cual se requeriría rellenar las tablas con los datos reales) sino sólo el encabezamiento de cada tabla asociada al que llamamos *esquema de la relación*.

[3]Se supondrá para simplificar que los nombres de los alumnos no pueden repetirse, y que también las asignaturas tienen nombres distintos.

Las relaciones establecidas para describir nuestro modelo no son las únicas posibles. De hecho, el planteamiento puede mejorarse fácilmente, sustituyendo la última relación por las dos siguientes:

Asignaturas (nombreAs, curso, númCréditos)

Matrículas' (nombreAl, nombreAs, nota)

en las que ya no se repiten el curso y número de créditos de cada asignatura por cada alumno que se matricule en ella, mejorando el aprovechamiento de la memoria y la gestión posterior de ella en consultas y actualizaciones de la información.

Este proceso (llamado *normalización*) no siempre resulta tan sencillo. Sin embargo, mediante su empleo se llega a describir una base de datos gracias a relaciones naturales y sin redundancias, con las ventajas mencionadas.

6.4 Lenguajes asociados a los SGBD

Visto el SGBD como un intermediario, deberá proporcionar al administrador un medio con que *definir* la base de datos durante su creación o realizar posteriores reestructuraciones, de acuerdo con los nuevos requerimientos, facilitándole el establecimiento de relaciones entre los datos. Finalmente, deberá poner al alcance del usuario toda la información para que éste la *manipule* de un modo flexible y eficiente.

Los SGBD proporcionan lenguajes de alto nivel para que los usuarios soliciten sus operaciones, ya que la propia base de datos oculta muchos detalles de la representación y el almacenamiento de las relaciones, y dichos lenguajes pueden obviar esos detalles.

Los lenguajes de los SGBD son en general de dos tipos:

- *Lenguajes de definición de datos*

  Estos lenguajes recogen la descripción de los datos tal como la concibe el usuario, esto es, la definición de los esquemas de la base

de datos. Como interpretación de esas definiciones, producen el diccionario de datos (véase 6.5) así como la forma real, a bajo nivel, en que los datos se organizan en los distintos dispositivos físicos, discos normalmente.

- *Lenguajes de manipulación de datos*

  Estos lenguajes permiten al usuario realizar operaciones tales como añadir información nueva en la base de datos, suprimir, modificar o consultar la existente.

  Estas operaciones tienen una gran potencia, ya que los lenguajes de consulta modernos disponen de mecanismos para definir las operaciones más frecuentes. Por otra parte, tienen también una gran flexibilidad, estando dotados de medios para que el usuario pueda desarrollar programas de acuerdo con sus necesidades, ya sea ofreciendo lenguajes de programación propios o bien enlazando con lenguajes de programación de alto nivel.

6.4.1 Lenguajes relacionales

Se han desarrollado dos enfoques de los lenguajes relacionales: el *álgebra relacional* y el *cálculo relacional*, aunque la mayoría de los lenguajes de consulta comercializados actualmente incluyen características de ambos. Entre ellos, es obligado citar el *SQL* (*Structured Query Language*), que funciona tanto en bases de datos desarrolladas para microcomputadores, como en otras para minis y mainframes, tales como Informix-SQL, Ingres y Oracle. A continuación presentamos brevemente los enfoques mencionados en el estilo de SQL.

- En los lenguajes basados en el *álgebra relacional* las operaciones se realizan con tablas y su resultado es también una tabla. Las principales son las siguientes:

  – La operación de *selección* extrae de una relación las tuplas que verifican cierto predicado. Por ejemplo:

  selección (Matrículas', nota $\geq$ 5)

consiste en la relación de las papeletas de aprobado; en cada una se consigna el nombre de un alumno, una asignatura y la nota obtenida. El esquema de la relación obtenida es

<nombreAl, nombreAs, nota, convocatoria>

que coincide con el de la relación *Matrículas'*.

- La *proyección* extrae de una relación las columnas que se indiquen, eliminando las filas repetidas en la relación resultante. Por ejemplo:

  proyección (Matrículas, <nombreAs, curso, númCréditos>)

  coincide con la relación de *Asignaturas*.

- La *concatenación* de dos relaciones que tengan alguna columna en común consiste en otra tabla, que pivota en la(s) columna(s) común(es). Por ejemplo, la relación *Matrículas* podría obtenerse así:

  concatenación (Matrículas', nombreAs)

Considerando que una tabla es un conjunto de tuplas, el álgebra relacional también incluye las operaciones conocidas de unión, intersección y diferencia (de conjuntos de tuplas), entre relaciones con la misma estructura.

- Los lenguajes basados en el *cálculo relacional* son declarativos: las consultas se expresan especificando la información deseada mediante el cálculo de predicados de primer orden. Por ejemplo, la definición siguiente

$$\{< n, d > \quad | \quad \exists\, t\,((< n, d, t > \in \mathit{Direcciones}) \wedge \forall\, a, c((< n, a, c > \in \mathit{Matrículas'}) \Rightarrow (c \geq 9))\}$$

facilitaría la relación de aquellos alumnos que han obtenido un sobresaliente en todas las asignaturas de las que se han matriculado, junto con su dirección.

6.5 Elementos de un SGBD

Siguiendo la presentación por capas que se hizo de los sistemas operativos, en una base de datos pueden distinguirse tres niveles:

- El más bajo se encuentra directamente sobre el sistema operativo, y consiste en los archivos de datos, los de índices y el diccionario de datos:
  - Los *archivos de índices* tienen idéntico cometido al del índice de un libro, permitiendo localizar rápidamente un registro a través de su dirección (página) en un archivo de datos (libro).
  - El *diccionario de datos* es un archivo que contiene la descripción de la estructura de los archivos de datos a los tres niveles; esto es, de los distintos esquemas de la base de datos.
- El nivel intermedio es el del *sistema de gestión* propiamente dicho, y comprende los siguientes módulos:
  - Un programa *compilador* para el lenguaje *de definición* de datos, que construye (o modifica) el diccionario de datos.
  - Un *intérprete* del lenguaje *de consultas*: cuando las instrucciones del lenguaje de manipulación de datos se insertan entre sentencias de otros lenguajes de alto nivel, el *precompilador* traduce esas instrucciones al lenguaje de alto nivel.
  - Un programa *gestor* de la base de datos, que es el intermediario entre los archivos, que contienen información a bajo nivel estructurada conforme al diccionario de datos, y los programas que acceden a ellos a un nivel más alto.
  - Es corriente que los SGBD faciliten rutinas de reorganización y generación de índices de los registros de la base de datos siguiendo ciertos criterios.
  - Frecuentemente se incluye un optimizador que, basándose en las definiciones de las tablas y los índices, crea el camino más eficiente de acceso a los datos. Este camino se construye en un paso de la compilación de un programa de alto nivel o en

el momento de la ejecución. Este módulo permite al usuario desentenderse de los detalles de nivel físico.

  - Muchos SGBD ofrecen también diversas rutinas de generación de programas, tales como formatos de pantalla, generación de etiquetas, listados o informes a medida.
  - Otro aspecto de gran desarrollo en la actualidad son las llamadas herramientas CASE[4], cuyo objetivo es facilitar el desarrollo y mantenimiento de programas de cierta envergadura.

- En el nivel más alto se encuentran el administrador y los usuarios de la misma:

  - El *administrador* de la base de datos es la persona que redefine el esquema de la base de datos en las sucesivas modificaciones. Además, concede o deniega a los distintos usuarios el acceso a toda la base de datos o a parte de ella.
  - Los *usuarios* pueden tener diferentes niveles de especialización, con diferentes grados de acceso a la base de datos.

    Si tomamos como ejemplo la base de datos de un banco, el nivel menos especializado lo representa un cliente, capaz de consultar los datos de su cuenta o modificarlos como consecuencia de una transacción efectuada en un cajero automático. En un nivel intermedio se hallarían los empleados, autorizados a realizar consultas y movimientos más complejos. Los programadores de aplicaciones se hallarían en el nivel más especializado.

6.6 Ejercicios

1. Los campos de un registro de un archivo pueden definirse mediante su tamaño (máximo), es decir, su número de caracteres o de dígitos enteros y decimales. En este supuesto, fije el tamaño que tienen los campos correspondientes a los siguientes datos: nombre, apellidos, domicilio, población, provincia, código postal y teléfono.

[4]Computer Aided Software Engineering: ingeniería de la programación asistida por computador.

2. Se desea gestionar una *videoteca* mediante una base de datos, manteniendo datos sobre los clientes y las cintas. Se considera la posibilidad de consultar la base de datos para conocer en un momento dado las existencias por temas, autores y actores.

   - Describa los niveles de visión y conceptual según las normas del ANSI/SPARC.
   - Identifique unos atributos apropiados para caracterizar las entidades "clientes" y "cintas".
   - Identifique las relaciones 1:1, 1:m y m:m que existan.
   - Construya un diagrama E-R.
   - Construya el esquema de relación correspondiente.

3. Encuentre un sistema de gestión de bases de datos e identifique sus características, valorando en qué medida alcanza los objetivos propuestos por los SGBD, cuáles son los elementos que posee y sus limitaciones.

4. En la base de datos referida a una carrera universitaria utilizada como ejemplo del modelo relacional se define la siguiente relación

   $$\text{Notas} \equiv (\text{nombreAl, curso, grupo, nombreP, nombreAs, feb, mayo, final})$$

   de cada alumno, en cada asignatura, con sus calificaciones de febrero, mayo y final. Se ha aprovechado la tabla para incluir el nombre del profesor que imparte la asignatura en ese grupo.

   - Mejore el planteamiento para evitar que se repita el nombre del profesor cada vez que aparezca un alumno suyo.
   - Evite asimismo la repetición del curso y grupo de cada alumno cada vez que aparece una asignatura.

5. Actuando como administrador de la base de datos del ejercicio anterior, establezca a qué información pueden acceder: un alumno, un profesor, el personal de administración, el jefe de estudios y el director del centro.

6.7 Comentarios bibliográficos

Las bases de datos constituyen, ya se ha dicho, un área de directa aplicación en muy diversos campos del mundo empresarial. Pero tampoco puede negarse

la contribución de los SGA, que también han evolucionado considerablemente, ofreciendo un elevado nivel, muy cercano a los problemas reales, proporcionando un ambiente sencillo, apto para su uso por no profesionales de la informática. Un ejemplo de SGA es el dBase, cuyo reducido coste, sencillo manejo y escasos requerimientos físicos la hacen tremendamente asequible. Entre los numerosos libros que se han escrito sobre ella, encontramos en [Bye90] una referencia básica y práctica de la versión *III plus*.

Entre las bases de datos relacionales que funcionan en PC's bajo DOS, deben mencionarse también Oracle y DB2/2. Aunque su medio principal son los entornos grandes, es posible usarlas en microcomputadores con fines educativos (lo que contribuye por otra parte a su difusión), así como en la fase de desarrollo. En [MW83] se introducen informalmente los conceptos básicos de las bases de datos relacionales, revisándose igualmente algunas de amplia difusión.

Posiblemente, SQL es el lenguaje de consulta de bases de datos más extendido en la actualidad (DB2, Oracle, Informix, Arity, RDB, etc.) En [HH89] se presenta este lenguaje, desde su fundamentación en la lógica del primer orden hasta su utilización en algunos SGBD comercializados.

Entre los numerosos textos sobre esta temática, indicamos [KS93], [Dat93] y [MP93], muy completos y actualizados.

En los años noventa ha surgido un nuevo enfoque de bases de datos, el de las orientadas al objeto, que está despertando un vivo interés y tiene una gran y rápida acogida. [SH90] es un estudio comparado de sus principios con los del modelo relacional.

Capítulo 7

Historia de los instrumentos de cálculo

El nacimiento de la Computación, tal como la vemos hoy día, es recientísimo (apenas tiene medio siglo de edad), e incluso está en pleno desarrollo; por eso, junto al interés que despierta el origen de los instrumentos de cálculo y la informática, surge la necesidad de conocer, siquiera someramente, su alcance en la actualidad, así como la de imaginar las expectativas de futuro previsibles.

7.1 Precursores de los computadores digitales

7.1.1 La antigüedad

Desde que los hombres aprendieron a contar han necesitado apoyarse en el cálculo para manipular cantidades y buscar métodos para facilitarlo. Estos métodos fueron bastante rudimentarios hasta que los sabios

y astrónomos hindúes recogieron la herencia greco-babilónica inventando la numeración posicional actual.

Con el desarrollo de esos métodos, fue necesario anotar números (ya fueran considerados datos, cantidades auxiliares o resultados finales), apareciendo así los primeros instrumentos de cómputo: el uso de los dedos con tal objeto condujo a las primeras técnicas de cálculo *digital*; y parecida finalidad tuvo el uso de piedras (*calculi*, término latino del que procede la palabra cálculo) entre los romanos, el de los nudos (*quipa*) sobre una cuerda, o las muescas practicadas por diversos pueblos sobre varas de madera.

Desde entonces, los avances se han debido a progresos en los métodos, en los instrumentos y en la integración de ambos.

Dos utensilios de esta época merecen ser citados: la máquina analógica de Anticíteros, precursora de los calendarios astronómicos bizantinos; y especialmente el *ábaco*, que apareció entre el tercer y el primer milenio a. C., y puede ser considerado como la primera máquina digital, ya basada en la numeración posicional.

7.1.2 Antecedentes del cálculo mecánico

Los autómatas de reloj que todavía adornan los campanarios de las iglesias medievales (a partir de finales del s. XIII) son automatismos mecánicos diseñados para reproducir una secuencia fija de movimientos. Precisamente se les ha llamado autómatas *de programa interior* para expresar así que la serie de sus movimientos está intrínsecamente descrita en su maquinaria.

7.1.3 La máquina de Pascal

Hacia la mitad del s. XVII, Pascal construyó una calculadora, capaz de sumar y restar, basándose en la pieza fundamental de los engranajes de los molinos: la rueda dentada. El funcionamiento de la máquina de Pascal puede compararse al de un cuentakilómetros: cada rueda posee diez posiciones y, a cada vuelta, provoca en la siguiente un arrastre de una posición.

Son directos descendientes de la máquina de Pascal las máquinas electromecánicas de oficina, hoy ya en desuso, y las UAL de los actuales computadores, cambiando las ruedas dentadas por circuitos electrónicos.

En 1673, Leibnitz perfeccionó la máquina de Pascal, incorporándole la multiplicación (por adición repetida del multiplicando en diferentes órdenes decimales) y la división.

Las rudimentarias operaciones que facilitaban estos aparatosos instrumentos no compensaban su costosa construcción. Por ello, los progresos consistieron en perfeccionamientos menores hasta finales del siglo XIX, en que se supo incorporar a las máquinas de cálculo una aportación procedente de la industria textil.

7.1.4 La máquina de Babbage

Las máquinas de Pascal y Leibnitz se consideran *de programa exterior*, debido a que las instrucciones son aportadas por el hombre junto con los datos, como ocurre con las calculadoras más simples. No obstante, sus instrucciones son demasiado simples como para poder considerarse un programa. Esta noción apareció cabalmente con la máquina de Jacquard, mecánico francés que en 1790 concibió la idea de un telar automático, capaz de tejer sus dibujos obedeciendo secuencias de instrucciones previamente registradas en tarjetas perforadas. El telar de Jacquard fue construido en 1800, y ha revolucionado la industria textil, pero el interés que tiene para nosotros es la aportación del verdadero concepto de *programa exterior*, tal como lo entendemos actualmente.

En el s. XIX, C. Babbage concibió su *Máquina Analítica* (1834), capaz de seguir las secuencias de instrucciones proporcionadas desde el exterior (como los datos), previamente registradas en tarjetas perforadas. El modus operandi de la máquina de Babbage consistía en ir leyendo tarjetas (descriptoras de operaciones) y ejecutándolas sucesivamente.

La gran aportación de Babbage consiste en reunir las dos siguientes características:

- la flexibilidad de las calculadoras de Pascal y Leibnitz, que operan sobre distintos datos, pero carecen de automatismo, estando su

velocidad siempre limitada por la lentitud de introducir los datos y las operaciones

- el automatismo de los autómatas de reloj, que no requieren la intervención humana, por lo que funcionan sin interrupción, aunque carecen de flexibilidad, ya que cada secuencia de movimientos requiere el diseño de una máquina distinta.

Babbage no pudo terminar la construcción de su máquina de diferencias. Repetidamente se ha sostenido que la complejidad de esta máquina rozaba probablemente los límites de la mecánica; sin embargo, estudios recientes [Swa93] demuestran que su diseño era completamente viable.

En el diseño de la máquina de diferencias se definen los órganos esenciales de cualquier sistema computacional actual:

Entrada	de instrucciones y datos
Almacén	memoria
Unidad de Control	con su actual cometido
Molino	calculador, U.A.L.
Salida	con su actual cometido

Por ello se le ha llamado el "Padre de la Computación moderna".[1]

7.1.5 La tabulación mecánica

El desarrollo de los estados modernos fue lo que determinó la necesidad de procesar grandes volúmenes de datos: siguiendo un mandato constitucional en EE.UU., era necesario realizar un censo de la población cada diez años. En 1886 se hizo patente la imposibilidad de obtener los resultados del censo de 1880 antes de 1890. Consciente de esta situación, Hollerit, funcionario de la oficina de censos, ideó un sistema de tabulación de los datos basado en tarjetas perforadas similares a las del telar de Jacquard.

[1]Entre los muchos campos que abarcó Babbage citamos los primeros métodos que posteriormente originaron la Investigación Operativa.

Con su sistema, Hollerit consiguió procesar el censo de 1890 en la cuarta parte del tiempo requerido para el de 1880. Su método fue tan eficiente que abandonó la Oficina de Censos, e inauguró una compañía de desarrollo de máquinas electromagnéticas (incorporando a su invento los avances en electromecánica y electricidad, construidos para perfeccionar el teléfono) que fue la precursora de la actual IBM.

Con el desarrollo de los computadores, las tarjetas perforadas pasaron a ser uno de los principales soportes de la información, situación que perduró hasta finales de los años setenta.

7.2 Nacimiento y evolución de los computadores digitales

En la primera mitad del siglo XX, gracias a los avances en tecnología electrónica, se crearon las primeras calculadoras electromecánicas, que funcionaban a base de relés.

Al poseer los relés dos estados (abierto y cerrado), constituyen el elemento ideal para representar los dos dígitos de la numeración binaria (0 y 1), y también los dos valores lógicos (verdadero y falso) del álgebra de Boole. Por ello, los relés pueden considerarse elementos de memoria capaces de albergar resultados parciales. Además, estos instrumentos eran capaces de seguir secuencias de instrucciones almacenadas previamente sobre tarjetas perforadas.

Los primeros computadores que funcionaron a base de tecnología eléctrica fueron:

1. En 1943, el Harvard Mark I, con relés electromagnéticos.

2. En 1943, el Colosus I, considerado como el primer computador electrónico, a base de (200) válvulas, diseñado durante la II Guerra Mundial para descifrar los mensajes del encriptador alemán Enigma.

3. En 1946, el ENIAC,[2] también a base de (18.000) válvulas, diseñado para confeccionar tablas balísticas.

A partir de los relés, la lenta mecánica fue poco a poco desplazada por la electricidad: el tubo de rayos catódicos, el transistor, los circuitos impresos y más tarde integrados, etc.

En realidad, estas máquinas estaban programadas al más bajo nivel (en términos de direcciones físicas de memoria, etc.) para desempeñar tareas muy específicas. Por ello, carecían por completo de versatilidad, debido a la dificultad que entrañaba reprogramarlas.

7.2.1 El modelo de von Neumann

En 1945, von Neumann dio el paso definitivo, incorporando a las máquinas de relés de entonces los dos siguientes conceptos:

1. *Programa registrado*, utilizando la memoria del calculador para almacenar el programa de instrucciones junto con los datos.

2. *Ruptura* (condicional) *de secuencia*, es decir, capacidad automática de decisión: dependiendo de cierto valor se ejecutaría una parte del programa u otra.

En 1945, von Neumann definió su modelo de computador,[3] que es en esencia el que funciona en nuestros días, con dos características:

1. La *secuencialidad* en la transmisión y tratamiento de los datos

2. La *codificación* de instrucciones mediante impulsos electromagnéticos, reemplazando el cableado

7.2.2 Generaciones tecnológicas

Las máquinas basadas en el modelo de von Neumann se comercializaron en los años cincuenta (1952-55) y, desde entonces, la carrera ha sido y sigue siendo imparable. De esta década parten las siguientes generaciones tecnológicas:

[2]Electronic Numerical Integrator And Calculator.

[3]También conocido como Máquinas de von Neumann.

Primera generación

Las válvulas electrónicas desplazaron a los relés, y los computadores adquirieron velocidad y potencia. Citamos entre los computadores de la primera generación el ENIAC, terminado en 1946, y el UNIVAC,[4] primera máquina con la arquitectura de von Neumann. El modus operandi era el siguiente:

- carga del programa y los datos, previamente perforados (T_1)
- ejecución (T_2)
- salida de los resultados (T_3)

El tiempo total invertido es la suma $T_1 + T_2 + T_3$. Las fases primera y tercera se llaman *tiempos de ocio* del procesador, porque en ellos permanecía inactivo.

Segunda generación

El aumento de potencia se debe a una innovación de los laboratorios Bell en 1948: el transistor. En cuanto al modo de trabajar, se introdujo la simultaneidad de las operaciones de cálculo con las de carga y salida, reduciéndose los períodos de ocio del procesador.

Para aumentar la velocidad de las operaciones de carga/salida, que retrasaban todo el proceso, se introdujo el uso de cintas magnéticas, más rápidas que las tarjetas perforadas. Este modo de trabajar se llama *off-lining*.

Tercera generación

Tecnológicamente, los avances en velocidad y potencia se deben a la incorporación de circuitos integrados (1952) y memorias de semiconductores (1971).

[4] UNIVersal Automatic Computer.

En esta generación el trabajo se caracteriza por explotar la simultaneidad del procesamiento con las operaciones de entrada/salida mediante la *multiprogramación*, consistente en que varios programas coexisten en memoria, y cuando uno de ellos debe realizar una operación de entrada/salida, el procesador no se detiene, sino que se dedica a otro programa. Así, la UCP nunca está ociosa habiendo trabajo por hacer, facilitándose además la asignación de prioridades a los programas que se encuentran a la espera.

Cuarta generación

Se incorporan nuevas tecnologías de fabricación y de integración de los componentes físicos entre sí (VLSI: *Very Large Scale of Integration*).

En cuanto a la explotación, se generalizan:

- el *teleprocesamiento*: extensión del sistema de carga y proceso a terminales remotas, sacando partido de la asignación de prioridades.
- los sistemas *interactivos* o *conversacionales*: los usuarios intervienen en el desarrollo de las sucesivas etapas, lo que trae consigo una mayor flexibilidad en la modificación y puesta a punto de programas.
- el *tiempo compartido*, con lo que se mejora el tiempo medio de respuesta.
- esta generación se corresponde cronológicamente con la difusión de los llamados *computadores personales*.

7.3 Evolución de los lenguajes y de la metodología

Mientras tanto, a partir de los años cincuenta, los lenguajes también emprenden la carrera de su evolución particular: para evitar la codificación de programas a bajo nivel, es preciso que sea la máquina la que adquiera lenguajes próximos al humano.

A la aparición del FORTRAN (FORmula TRANslator system) siguieron el Algol y el Cobol, intentando acercarse a un lenguaje universal. La avalancha posterior de lenguajes y dialectos demostró que esas tentativas de universalidad resultaron vanas.

Por otra parte, la actividad de la programación experimentó en los años sesenta un desarrollo mucho más rápido que las técnicas empleadas en él; esta crítica situación se conoce como *crisis del software*. No existía una metodología de la programación: cada programador desarrollaba sus propios métodos o "trucos" para programar, y se consideraba a la programación más como un arte que como una técnica. Muy pronto surgió la necesidad de sistematizar la programación, desarrollando herramientas y hábitos de diseño metódicos y disciplinados para conseguir que los programas fueran correctos, eficientes y mantenibles. La tesis de la programación estructurada y las aportaciones de Dahl, Dijkstra, Hoare [DDH72] y el propio Wirth, vinieron a conferir a la programación un carácter de disciplina.

Aparece así la *programación estructurada y modular* y comienzan a aplicarse técnicas de diseño de algoritmos como el *método de los refinamientos sucesivos*. En este contexto nace el lenguaje Pascal, que fue creado con el propósito de enseñar a programar de una forma metódica y disciplinada. De hecho, se considera como un lenguaje ideal para entrar en contacto con la programación, por lo que es el primero que muchas universidades vienen enseñando durante años. Los nuevos lenguajes de programación que aparecen a partir de los años setenta incorporan estas técnicas, facilitando su desarrollo y aplicación.

Posteriormente, los lenguajes de programación han seguido evolucionando, y paralelamente las técnicas de desarrollo de algoritmos. En el

capítulo 5 se han recogido algunos de los modelos de programación más empleados.

7.4 Tecnología actual, tendencias y perspectivas

A partir de la aparición del computador personal al principio de los años ochenta, la informática ha dejado de ser un coto cerrado de un grupo restringido de especialistas y se ha popularizado, alcanzando a usuarios sin ningún tipo de conocimientos previos en este área.

Hoy en día, la Informática está difundida por un sinfín de sectores: las finanzas, la gestión, la investigación, el control de procesos industriales (diseño, concepción y fabricación asistidas por computador), la educación, la edición, la composición musical, etc., y el número de usuarios que utilizan la informática en su trabajo o en su ocio sigue aumentando.

En los países desarrollados, cada día está más cercana la predicción de "un computador en cada puesto de trabajo y en cada hogar".

Por ello, se hace necesario facilitar la relación entre el usuario y el computador y disminuir el tiempo de aprendizaje, y para lograrlo se recurre a modelos más intuitivos que muestran al computador como un escritorio o mesa de trabajo, a las aplicaciones como tareas y los ficheros de datos como carpetas de documentos, mediante entornos de tipo gráfico (4.2.5).

A su vez, se ha dado una notable evolución tecnológica: los computadores son cada vez más potentes y más baratos. Se puede afirmar que esta evolución es más rápida en el campo del hardware que en el software. Aparecen computadores con enormes posibilidades, pero los programas capaces de aprovecharlas llegan con varios años de retraso.

La mayor capacidad de proceso está permitiendo la entrada de la informática en campos en los que su presencia era limitada, y la creación de actividades enteramente nuevas, como son:

1. Visión artificial: reconocimiento de formas

2. Tratamiento de imágenes: corrección de aberraciones y defectos ópticos.

3. Realidad virtual: síntesis de imágenes virtuales estereoscópicas, películas virtuales.

4. Comunicaciones: enlaces mediante módem, correo electrónico, establecimiento de redes integradas de comunicaciones (datos, sonido y vídeo).

5. Juegos: de rol, arcade, ajedrez.

6. Simulación de procesos

7. Multimedia: procesamiento de datos, sonido y vídeo integrados.

De hecho, todo el mundo coincide en calificar este enorme desarrollo de la informática como una verdadera revolución de la información. Como tal revolución, tiene importantes repercusiones de índole económica y social. La utilización de la informática genera un aumento de la productividad y de la calidad de la producción. En consecuencia, las empresas se ven obligadas a informatizarse para poder competir en el mercado. El aumento de la productividad puede significar también un aumento del desempleo o del ocio.

Para imaginar cómo serán los computadores del futuro, quizás lo más razonable sea identificar las limitaciones de los computadores actuales, e imaginar sus formas de superación.

Por lo pronto, la naturaleza física de los componentes electrónicos los limita en tamaño y en velocidad. De ahí que la investigación se oriente hacia la superación de las configuraciones tradicionales (basadas en el modelo de von Neumann) por otras arquitecturas y redes más avanzadas (paralelas, neuronales, ...) llamadas a veces "arquitecturas de flujo de datos" (véase el capítulo 3). En ellas, un programa no se compone de instrucciones que se ejecutan "de arriba abajo", sino de segmentos que pueden "resolverse" en cuanto los datos precisos estén disponibles. Los datos fluyen a través de complicadas redes, compitiendo por acaparar la

atención de los recursos (principalmente los procesadores, que trabajan simultáneamente).

7.4.1 Inteligencia artificial

El ambicioso objetivo que se persigue es dotar a los computadores del futuro con una serie de capacidades que se engloban bajo el término genérico de Inteligencia Artificial. Esta disciplina se ocupa en la actualidad de las siguientes áreas de investigación:

- lenguajes naturales: síntesis del habla, identificación del lenguaje hablado, traducción automática
- razonamiento y aprendizaje automáticos, sistemas expertos, demostración automática
- programación automática

Todas esas capacidades requieren, en general, un mayor estudio del proceso cognitivo. En particular, todas ellas necesitan sustituir el empleo de datos por el de conocimientos, lo que plantea el difícil problema de su representación.

7.4.2 Las comunicaciones

Otro aspecto de la informática con tremendo auge en la actualidad es el de las comunicaciones, ya que se abre la posibilidad de acceder a la información de origen remoto, tratarla automáticamente y enviar los resultados a su lugar de origen.

Para ello se crean redes de comunicaciones de mayor o menor cobertura, desde una sala o un edificio hasta redes nacionales o mundiales.

Las aplicaciones de la *teleinformática* están cada día más extendidas y ya casi empiezan a parecernos algo natural. Como ejemplos podemos citar desde el uso de los cajeros automáticos hasta la posibilidad de efectuar reservas de avión o de hotel desde puntos remotos, la consulta del catálogo de una biblioteca en una ciudad de otro país, el correo electrónico, etc.

7.5 Comentarios bibliográficos

[JM90] y [Ber86] son dos buenas recopilaciones, con carácter general, sobre la historia pasada de los instrumentos de cálculo, desde la calculadora de Pascal hasta nuestros días. [Ber86] es extraordinariamente entretenido, con un buen número de anécdotas y curiosidades sobre los padres de la computación moderna. Especialmente interesantes son las citas de escritos de von Neumann recogidas en él.

En [Swa93] se presentan los avatares en que se vio envuelta la construcción de la máquina analítica de Babbage, demostrando que el proyecto era viable, a pesar de no verse finalizado: En este artículo se describe la construcción de la máquina analítica con motivo del segundo centenario del nacimiento de Charles Babbage, y de acuerdo con sus planos originales.

Aunque los computadores digitales actuales tienen su origen en el modelo de von Neumann (1945), la invención del primer computador electrónico es anterior, y se debe a J. V. Atanasoff, que construyó el primer computador digital entre 1937 y 1942. De hecho, es posible que los creadoes del ENIAC copiaran algunas de sus ideas ([Mac88]).

Sobre computación avanzada en la actualidad y perspectivas de futuro de la informática pueden consultarse [Com87] y [Sim85]. [La 91] trata sobre este mismo tema, en relación con las comunicaciones.

Apéndice A

Introducción al DOS

El DOS es, hoy en día, *el* sistema operativo más difundido para computadores PC de IBM y compatibles. Dada la gran difusión de éstos, se comprende la importancia que tiene conocer el DOS.

Aunque los PCs pueden comunicarse con otros a través de redes, su modo habitual de trabajar es autónomo, por lo que el DOS es un sistema operativo *monousuario*. Su cometido (véase el capítulo 4) consiste en gestionar los recursos del sistema, controlando el uso de los dispositivos (como las unidades de disco, la impresora, el ratón), administrando la memoria principal y facilitando la ejecución de programas y la organización de archivos dentro de los discos.

El DOS es en realidad un programa (o mejor dicho un conjunto de programas) grabado en un disco.[1] Cuando se enciende un equipo, se empieza cargando este programa en la memoria principal y se efectúan las tareas de puesta en marcha (véase la sección A.3); entonces entra en funcionamiento un intérprete de mandatos, que espera las órdenes del usuario (véase la sección A.2), y las va ejecutando sucesivamente.

Asumimos en adelante una configuración con disco fijo (duro) en el que se encuentra este programa de arranque del DOS, que es actualmente el caso más habitual.

[1] De hecho, ése es el significado de las siglas que forman su nombre: "Disk Operating System", que quiere decir en inglés Sistema Operativo de Disco.

A.1 Organización de recursos

A.1.1 Principales dispositivos

Durante el trabajo con el DOS, a veces necesitamos referirnos explícitamente a los periféricos; para ello, se emplean los siguientes términos en el lenguaje del DOS:

identificador	periférico
`nul:`	nulo
`con:`	consola = teclado + monitor
`a:` `b:` `c:` ...	unidades de disco
`lpt1:` = `prn:`	impresora
`lpt2:` ...	otros dispositivos en paralelo
`com1:` = `aux:`	dispositivo en serie
`com2:` ...	otros dispositivos en serie

Todos los dispositivos pueden nombrarse omitiendo los dos puntos finales, a excepción de las unidades de disco que deben llevarlos obligatoriamente.

A.1.2 Archivos

Un *archivo* del DOS no es más que un documento, aunque en vez de estar representado por una secuencia de letras escritas sobre papel, consiste en una secuencia de bytes (que representan caracteres igualmente) grabados sobre una zona de un disco magnético. Dependiendo del contenido de los archivos, desde el punto de vista del DOS, algunos archivos son "de datos" y otros son "programas", escritos en lenguaje de máquina o del DOS.

Los archivos se nombran mediante palabras, llamadas *identificadores*, formadas por letras, dígitos y los caracteres siguientes:

```
~ ! @ # $ % ^ & ( ) - _ { } '
```

Los identificadores de los archivos tienen dos partes: *nombre* (de un máximo de ocho letras) y *extensión* (de un máximo de tres), siendo esta última opcional. Al escribir el identificador de un archivo, estas dos partes se separan mediante un punto. Usando la notación EBNF (véase el capítulo 5), esta descripción puede expresarse así:

$$\textit{identificador} ::= \textit{nombre}[.\textit{extensión}]$$

Ateniéndonos a estas reglas, conviene elegir los nombres de los archivos relacionados con su contenido. En cuanto a las extensiones, su elección depende del tipo de archivo; las más usuales son las siguientes:

extensión	uso habitual
`com` `exe`	Programas ejecutables
`bat`	Lotes de órdenes (v. A.2.5)
`sys`	Programas del sistema
`bas` `pas` `cob`	Código fuente de programas en Basic, Pascal o Cobol
`txt`	Archivos de texto

Lógicamente, los identificadores usados para representar archivos no pueden coincidir con los identificadores propios de dispositivos.

Con frecuencia, se necesita realizar ciertas operaciones con un grupo de archivos más o menos grande. Si esos archivos fueron bautizados siguiendo ciertas pautas, es posible identificarlos genéricamente mediante un *patrón* o *plantilla* de archivo. Para ello, se usan los *caracteres comodín*, `*` y `?`, que representan respectivamente cualquier cadena de caracteres como terminación del nombre o extensión, y *un* carácter en cualquier posición del identificador. Por ejemplo:

Plantilla	Interpretación
`arch*.com`	`archivo.com, archi-1.com, arch.com,...`
`c9?-9?.txt`	`c92-93.txt, c93-94.txt, c9a-9f.txt,...`
`*.exe`	Todos los archivos con esa extensión
`*.*`	Todos los archivos

A.1.3 Directorios

El número de archivos grabados en un disco llega con frecuencia a ser enorme, por lo que se hace necesario clasificarlos. Éste es el cometido de los *directorios* y *subdirectorios*, algo así como compartimentos y compartimentos dentro de compartimentos que permiten organizar jerárquicamente los archivos de un disco.

En general, los identificadores de los directorios siguen las mismas reglas que los archivos, aunque en el caso de los directorios es bastante frecuente omitir la extensión. Además de los identificadores corrientes, existen símbolos especiales para indicar las siguientes posiciones:

identificador	posición
\	el directorio principal o raíz
.	el directorio actual
..	el directorio "padre"

La comparación de los directorios con compartimentos se refleja gráficamente en el ejemplo de la figura A.1, donde los identificadores de los directorios y subdirectorios señalan puertas (de entrada), y los archivos están representados directamente por sus identificadores. Otro modo más sencillo e igualmente útil de representarlos es como árboles, según se muestra en la figura A.2.

Es importante tener en cuenta que los archivos contenidos en cada unidad de trabajo están organizados en forma de árbol, por lo tanto, cada unidad tiene su propio directorio raíz y en todo momento cada unidad tiene un directorio actual.

Además, es posible referirse a cualquier directorio o conjunto de archivos indicando una ruta de acceso desde la posición actual, esto es, el directorio de trabajo[2] (véase 4.2.2). Con este fin, el símbolo \ sirve para concatenar los pasos hacia dicha posición.

Así por ejemplo, considerando que el disco `C:` está organizado según la estructura anterior, es posible referirse a los archivos `ejerci.tex` y `examen.tex` desde diversos lugares:

[2]Esta posición se conoce frecuentemente como directorio "por defecto", a causa de una traducción dudosa de su nombre *default directory*, en inglés.

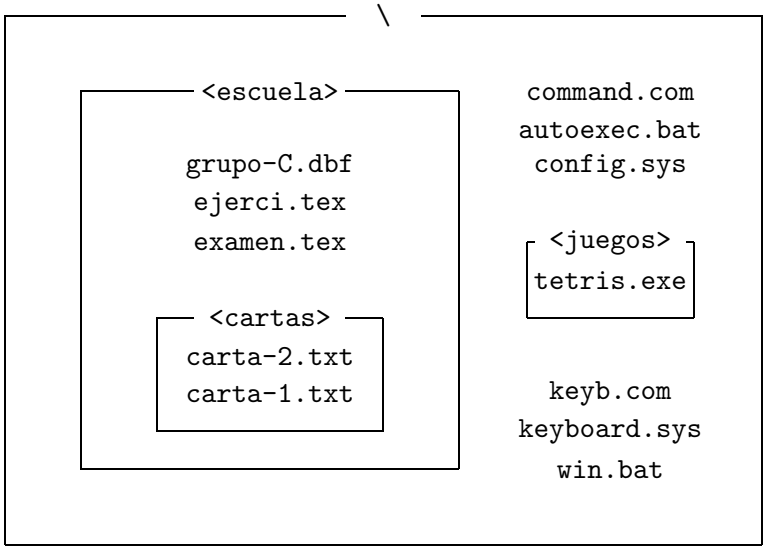


Figura A.1.

- Desde `C:\`

```
escuela\*.tex
```

- Desde `C:\escuela`

```
*.tex
```

- Desde `C:\escuela\cartas`

```
..\*.tex
```

- Desde `C:\escuela\juegos`

```
..\escuela\*.tex
```

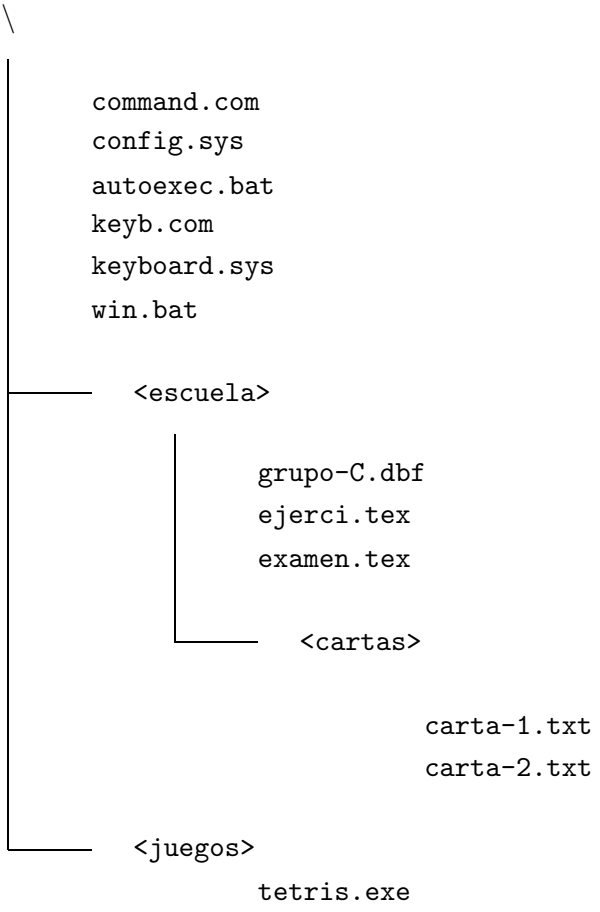

\
command.com
config.sys
autoexec.bat
keyb.com
keyboard.sys
win.bat
<escuela>
grupo-C.dbf
ejerci.tex
examen.tex
<cartas>
carta-1.txt
carta-2.txt
<juegos>
tetris.exe

Figura A.2.

- Desde cualquier lugar del disco `C:`

```
\escuela\*.tex
```

- Desde cualquier lugar, incluso desde otro disco:

```
C:\escuela\*.tex
```

A.1.4 Prompt

El *prompt*[3] es un indicador con el que el DOS expresa que está esperando nuestras órdenes, listo para ejecutarlas. El DOS puede indicar esta disposición de diversas formas, a nuestro capricho, pero la más extendida consiste en recordar el disco de trabajo y la posición, dentro de él, en ese momento; suele finalizarse con el carácter `>`. Por ejemplo, el prompt

```
C:\ESCUELA\CLASES>
```

indica que la posición de trabajo actual es el subdirectorio `CLASES`, del directorio `ESCUELA`, en el disco `C:`. A la derecha del prompt aparece el cursor, bajo nuestro control, para escribir la orden que deseemos.

A.2 Órdenes del DOS

El sistema DOS ofrece un intérprete de mandatos que permite al usuario la comunicación con el computador.

Empecemos por poner unos cuantos ejemplos de órdenes, junto con su funcionamiento. En adelante, usaremos letra de molde para reproducir el intercambio de información entre computador y usuario; lo escrito por el computador aparece en letra vertical y lo escrito por el usuario en letra oblicua:

[3] Aunque en español significa incitar o incitación, sólo usamos el término inglés, sin traducir.

```
C:\>CD pascal
C:\PASCAL>A:
A:\>Dir *.CHI

   El volumen en la unidad A no tiene etiqueta
   El número de serie del volumen es 335E-11EF
   Directorio de A:

  ENTROPY    CHI    1633    21/06/93      9:08
  REGLAMEN   CHI    2204    27/06/93     20:07
  JUNIO-93   CHI    3425    29/06/93     17:03
           3   Archivo(s)  176640 bytes libres

A:\>FORMAT B:
 Comando o archivo no se encuentra

A:\>C:\DOS\FORMAT B:
... ... ...
A:\>CLS
```

Debe advertirse que en el lenguaje del DOS no se distingue entre las letras mayúsculas y las minúsculas. Esto afecta a los identificadores (ya sea para dispositivos, archivos o directorios), así como a las órdenes que estudiaremos.

Por otra parte, los comandos del DOS pueden agruparse en dos categorías: *órdenes (internas)* del DOS y *programas (externos)*. El DOS conoce las primeras, y es capaz de interpretarlas y ejecutarlas en cualquier momento; también es posible ordenar la ejecución de un programa tras el prompt pero, para ejecutarlo, el DOS debe leer *el disco* para pasarlo a la memoria del computador. Por lo tanto, sólo es posible ejecutar programas grabados en un disco accesible por el DOS.

Por ejemplo: el cambio de unidad, las órdenes `CLS`, `DIR` y `CD` son órdenes internas; `FORMAT` en cambio es un programa. El interés de esta observación consiste en que, cuando se ordena un comando externo o programa, para que éste pueda ejecutarse deberá darse alguna de las siguientes circunstancias:

- el programa deberá estar en la unidad y directorio de trabajo actual, o

- se indicará su posición, para que el DOS pueda encontrarlo, o

- estará en alguna de las posiciones conocidas de antemano por el DOS en los que busca las órdenes de uso frecuente (véase la sección A.2.3).

En los siguientes apartados se describirán los comandos del DOS más usuales. Interesa saber de cada uno si es externo o interno, su propósito y su sintaxis correcta. Usaremos las reglas EBNF (introducidas en el capítulo 5) escribiendo los símbolos terminales con letra vertical y los no terminales con letra oblicua, en vez de distinguirlos con mayúsculas y minúsculas, o de usar los metasímbolos `<` y `>`, por tener éstos otro cometido.

Así por ejemplo, podemos decir que la estructura general de las órdenes del DOS es la siguiente:

comando [*argumentos*] [*opciones*]

donde la primera palabra es el nombre del comando, y aparece siempre; los argumentos son los objetos que maneja la orden, frecuentemente archivos o directorios, y no siempre aparecen (por ejemplo, la orden `CLS` no tiene argumentos); y las opciones son específicas de cada comando, como `/W`, y alteran el funcionamiento del mismo. En estas descripciones sintácticas aparecerán con frecuencia los siguientes símbolos no terminales:

- *Unidad*: `A:`, `B:`, `C:`, ...

- La [*posición*], opcional, se indica cuando procede, dando una unidad de disco y/o un camino de acceso, bien desde el directorio raíz (posición *absoluta*) o desde el (sub)directorio actual abierto en el disco correspondiente (posición *relativa*):

  [*unidad*][*camino*]

- *Archivo(s)*, que muchas veces se expresarán de forma genérica mediante una plantilla o patrón, usando los símbolos comodín.

- Las *opciones*, tales como `/B`, `/S`, ...

Por otra parte, muchas de las órdenes incluyen en su sintaxis partes opcionales, frecuentemente un disco, una posición dentro de él o un conjunto de archivos. Cuando esas partes opcionales se omiten, se asume que se trata del disco de trabajo actual, del (sub)directorio abierto, de todos los archivos a la vista, etc.

A.2.1 Órdenes básicas

`CLS` (interna)

- Propósito: Borra la pantalla, situando el prompt y el cursor en su comienzo.

- Sintaxis: `CLS`

Cambio de unidad (interna)

- Propósito: Establece una nueva unidad de trabajo.

- Sintaxis: *unidad*

`FORMAT` (externa)

- Propósito: Preparar un disco para ser usado por el DOS, creando el directorio (vacío) inicial.

- Sintaxis: [*posición*]`FORMAT` *unidad* [*opciones*]

- Algunas de las opciones usadas con este comando son las siguientes:

  - `/S`, para transferir a un disco el sistema operativo, haciendo así posible iniciar el funcionamiento del DOS con ese disco.
  - `/V`, para dar un nombre a un disco.

A.2.2 Manejo de archivos

`DIR` (interna)

- Propósito: Proporciona información sobre:
  - la posición especificada,
  - los archivos visibles en esa posición,
  - las puertas (de acceso a directorios) a la vista, incluyendo el subdirectorio actual, desde la posición indicada,
  - la memoria libre en una unidad.
- Sintaxis: `DIR` [*posición*][*archivo(s)*] [`/W`][`/P`]
- Ejemplo:

```
C:\PASCAL-6> DIR /w

   El volumen en la unidad C tiene etiqueta BLAISE
   El número de serie del volumen es 0F1E-18DF
   Directorio de C:\PASCAL-6

.            ..           TVISION     TVDEMOS
UTILS        TURBO3       TURBO   EXE UNZIP    EXE
README  COM  TURBO    TPL TURBO   TP  TPC      EXE
TPTOUR  EXE  TPTOUR   CBT TPTOUR1 CBT TPTOUR_P CBT
TPTOUR  CBT  TPTOUR_U CBT README      TURBO    HLP
TPC     CFG  NONAME00 PAS
      22 Archivo(s) 5171200 bytes libres
```

- Opciones
  - `/P` sirve para parar la relación de página en página.
  - Con la opción `/W`, sólo se da el identificador de cada archivo y se aprovecha la pantalla a lo ancho, con lo que la relación ocupa menos líneas.

`COPY` (interna)

- Propósito: Copia uno o varios archivos, con el mismo nombre o distinto; también se usa para transferir archivos hacia o desde un dispositivo.
- Sintaxis: `COPY` [*posición*][*archivo(s)*] [*posición*][*archivo(s)*]
- Este comando tiene dos argumentos: el(los) archivo(s) fuente y el(los) archivo(s) destino; sin embargo, es frecuente omitir uno de ellos. Por ejemplo, las dos órdenes siguientes

```
C:\> COPY autoexec.bat a:\
A:\> COPY c:\autoexec.bat
```

  tienen el mismo efecto.

`COMP` (externa)

- Propósito: Compara los contenidos de uno o varios pares de archivos.
- Sintaxis:

  [*posición*]`COMP` [*posición*][*archivo(s)*] [*posición*][*archivo(s)*]

`DEL` (interna)

- Propósito: Borra un archivo o un grupo de ellos de un disco.
- Sintaxis: (`ERASE` | `DEL`) [*posición*][*archivo(s)*]

`REN` (interna)

- Propósito: Renombra el identificador de un archivo o grupo de ellos.
- Sintaxis: (`REN` | `RENAME`) [*posición*][*archivo(s)*] [*archivo(s)*]

`TYPE` (interna)

- Propósito: Muestra en la pantalla el contenido de un archivo.
- Sintaxis: `TYPE` [*posición*][*archivo(s)*]

`PRINT` (externa)

- Propósito: Envía un archivo a la impresora, colocándolo en la cola de impresión.
- Sintaxis: [*posición*]`PRINT` [*posición*][*archivo(s)*] [`/C`][`/T`]
- Opciones
  - Con `/C` se cancela(n) de la cola el (los) archivo(s) especificados.
  - `/T` cancela toda la cola de impresión

A.2.3 Manejo de directorios

`CD` (interna)

- Propósito: Cambia el directorio de trabajo actual.
- Sintaxis: (`CHDIR` | `CD`) [*unidad*]*camino*

`TREE` (externa)

- Propósito: Muestra la estructura de subdirectorios de un directorio dado, de un modo muy similar al mostrado en la figura A.2.
- Sintaxis: `TREE` [*directorio*] [`/F`]
- Opciones
  - Con la opción `/F` se incluyen los archivos contenidos en cada subdirectorio.

`MD` (interna)

- Propósito: Construye un nuevo subdirectorio.

- Sintaxis: (`MKDIR` | `MD`) [*unidad*]*camino*

`RD` (interna)

- Propósito: Borra un subdirectorio, que debe estar vacío.
- Sintaxis: (`RMDIR` | `RD`) [*unidad*]*camino*

`PATH` (interna)

- Propósito: Establece rutas de búsqueda alternativas a la posición de trabajo actual.
- Sintaxis: `PATH` [*posición*];{*posición*}]

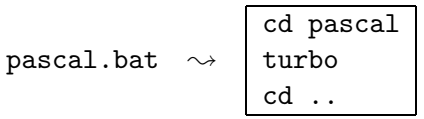
A.2.4 Indicadores del sistema

Durante el funcionamiento del DOS existen unos cuantos datos de referencia, tales como el path, a las que se llama *indicadores*:

- `Date` y `time` permiten conocer o alterar la fecha y la hora del sistema. Aunque su valor es actualizado constantemente por el propio computador, también el usuario puede establecer uno.
- `Ver` es la versión del DOS en uso.
- `Verify` es un conmutador (con dos posiciones: *on* y *off*) que, cuando está activo, indica al DOS que compruebe las operaciones de escritura de archivos.
- `Break` es también un conmutador: cuando está activo (*on*), el DOS comprueba frecuentemente si se producen interrupciones durante la ejecución de programas.
- El `prompt` ya ha sido introducido. Por lo general, su aspecto y la información que proporciona se establece durante el arranque del sistema (véase la sección A.3), aunque el usuario puede redefinirlo a su antojo haciendo que indique, por ejemplo, la hora del sistema.

A.2.5 Procesamiento por lotes

Cuando se necesita ejecutar repetidamente una secuencia de órdenes, es posible definir un *lote* de órdenes (véase 4.1.2) (por ejemplo, la secuencia `cd pascal`, `turbo` y cd ..), agrupándolas bajo un nombre (como `pascal.bat`, por ejemplo), de manera que baste con ordenar ese identificador para que el DOS lleve a cabo la secuencia, una a una:



El identificador asociado a un lote de órdenes debe tener la extensión `bat`[4] obligatoriamente. Sin embargo, en su posterior utilización puede omitirse la extensión.

Como un lote de órdenes es un archivo de texto, puede construirse con un editor cualquiera, o copiándolo directamente del teclado, mediante:

```
copy con:  pascal.bat
```

donde el retorno de carro representa el final de línea, y la combinación `Ctrl+Z` la marca del fin de archivo.

En un archivo `bat` pueden usarse, además de las órdenes usuales del DOS, otras específicas (`ECHO`, `REM`, `PAUSE`, `GOTO`, `IF`, `FOR`), con las que se pueden construir programas con gran flexibilidad.

A.3 Configuración del DOS

El elemento del sistema operativo de más bajo nivel es el BIOS: el sistema básico de entrada y salida (del inglés, *Basic Input/Output System*). Se trata de un conjunto de rutinas situadas en la memoria ROM del computador que realiza tareas básicas como las pruebas de que los periféricos están conectados o el programa para poder leer los ficheros del DOS, bien del disco duro o bien de discos flexibles. Estas rutinas no son suministradas por el fabricante del sistema operativo sino que son

[4]En inglés, *batch* significa lote.

entregadas con el hardware, por lo que se suele utilizar la expresión *firmware* para hacer referencia a este tipo de software introducido en la ROM.

Ya se ha dicho que el DOS está grabado en un disco. En efecto, cuando se enciende el computador, el programa de la ROM extrae del disco un programa generador del DOS que, en líneas generales, da los siguientes pasos, representados en la figura A.3:

1. Construye y carga en la memoria RAM la primera parte del sistema operativo, responsable de las operaciones de entrada y salida (*BIOS*), así como de la gestión de archivos y directorios y de la ejecución de los programas (*núcleo*).

2. Se busca en el disco un archivo llamado `CONFIG.SYS` y, si existe, se adoptan sus mandatos, estableciendo ciertos parámetros iniciales del DOS e instalando algunos controladores de dispositivos, tal como veremos a continuación.

3. Se carga en la memoria el intérprete de comandos; normalmente es el programa `COMMAND.COM`, aunque en el archivo `CONFIG.SYS` podría haberse especificado otro.

4. Finalmente, se busca en el disco el archivo `AUTOEXEC.BAT` y, si existe, se ejecuta este lote de órdenes.

Los archivos `CONFIG.SYS` y `AUTOEXEC.BAT` pueden ser modificados (o creados) por el usuario para configurar el funcionamiento del DOS según sus necesidades, y obtener de él el máximo rendimiento.

El siguiente ejemplo de archivo `CONFIG.SYS`

```
BREAK=ON                                          (1)
COUNTRY=34                                        (2)
BUFFERS=25,8                                      (3)
FILES=20                                          (4)
SHELL=C:\DOS\COMMAND.COM                          (5)
INSTALL=C:\DOS\KEYB.COM SP,,C:\KEYBOARD.SYS       (6)
```

opera así:

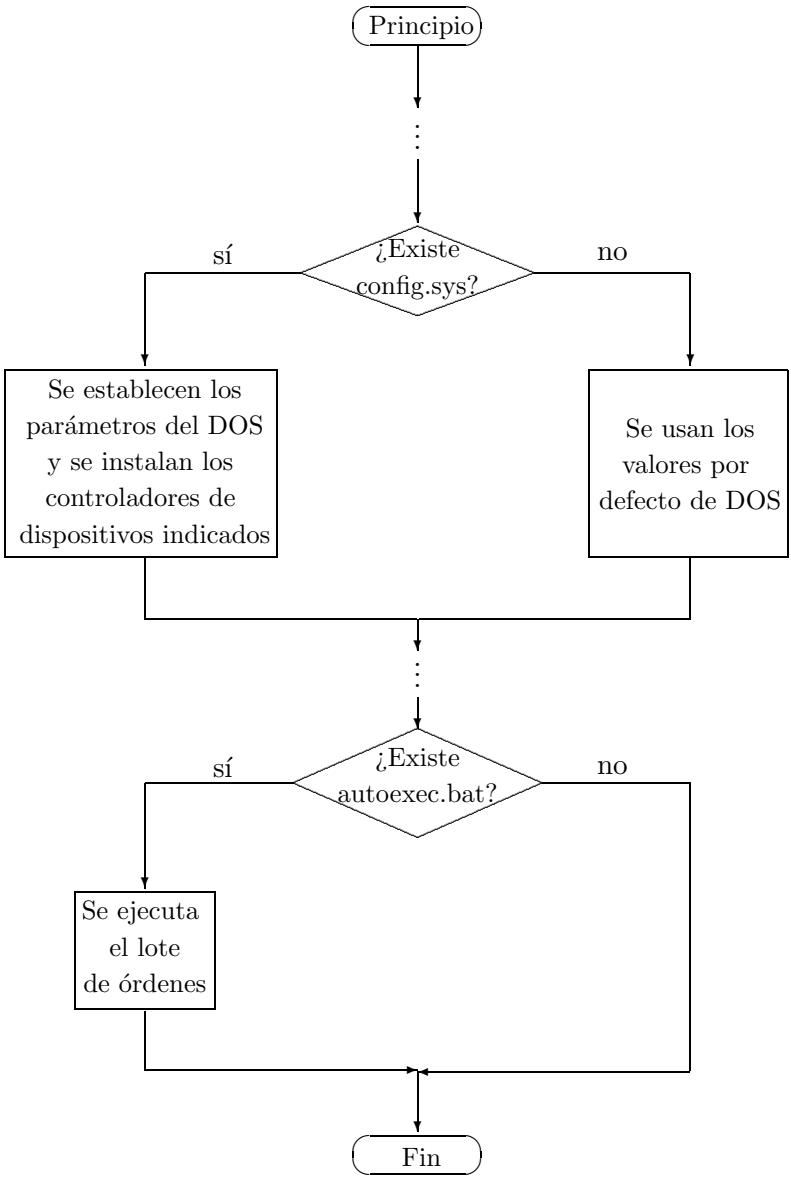


Figura A.3. Diagrama de flujo del arranque de DOS.

1. Activa la verificación de interrupciones.

2. Establece los formatos de fechas, horas, separadores y moneda del país.

3. Establece el tamaño de ciertas zonas de memoria usadas durante las operaciones de entrada y salida con el disco.

4. Establece el número máximo de archivos que es posible tener abiertos a la vez.

5. Elige el intérprete de comandos usual.

6. Finalmente, instala el programa controlador del teclado en castellano.

Terminamos esta sección con un ejemplo de `AUTOEXEC.BAT` típico, que no necesita comentario alguno:

```
CLS
DATE
TIME
ECHO  OFF
PATH  C:\;C:\dos;C:\LOTES;C:\UTILES;C:\PASCAL
PROMPT $P$G
VER
KEYB  SP,,C:\DOS\KEYBOARD.SYS
C:\RATON\MSMOUSE /2
INSTALL=C:\DOS\KEYB.COM  SP,,C:\KEYBOARD.SYS
SET EDITOR=C:\UTILES\epsilon
```

A.4 Otros aspectos de interés

A.4.1 Encauzamiento: tubos y demás

En principio, órdenes como `DIR` y `TYPE` dirigen su salida hacia la pantalla. El intérprete de comandos del DOS nos ofrece la posibilidad

de desviar la salida de esas órdenes hacia otro dispositivo, mediante los operadores de redireccionamiento:

```
...>DIR /w > lpt1:
```

o confeccionar un archivo con ella:

```
...>DIR /w > dir\list.txt
```

e incluso agregar esa información a la de un archivo ya existente:

```
...>DIR >> dir\list.txt
```

Otras órdenes toman su entrada del teclado:

```
...>DEL *.*
¡Se eliminarán todos los archivos del directorio!
¿Está usted seguro?  (S/N) S
...>
```

En cambio, si se tiene un archivo llamado `si.txt`, cuyo contenido consiste únicamente en el carácter "s":

`si.txt` ⇝ `s`

es posible conseguir que una orden capte su entrada del mismo, así:

```
...>DEL *.* < SI
...>
```

Los siguientes programas, llamados *filtros*, permiten sacar partido del redireccionamiento:

- `SORT` toma como entrada unas cuantas líneas del teclado o de un archivo de texto (donde el retorno de carro representa el final de línea y la combinación `Ctrl+Z` la marca del final), escribiendo a continuación las mismas líneas, pero en orden alfabético (ASCII):

  ```
  ...> SORT
  uno
  dos
  tres
  ```

```
^Z
dos
tres
uno
...>
```

- `MORE` escribe su entrada en la pantalla, de página en página.
- `FIND` busca en un archivo el texto especificado, mostrando todas las líneas que lo contengan:

```
...>FIND "ornitorrinco" < australi.txt
```

Al igual que se puede dirigir la salida mediante `>` y `>>`, es posible convertir un archivo en argumento de un programa:

```
...> SORT < agenda.txt

adela           2008586
bernardo        7388196
...
zutano          5494389
```

Otra interesante utilidad consiste en convertir una salida por la pantalla en el argumento de un programa:

```
...>DIR | SORT
```

pudiéndose combinar con el redireccionamiento:

```
...>DIR | SORT > DirOrden.txt
```

A.4.2 Atributos y protección de archivos

Los sistemas operativos ofrecen mecanismos de protección de la información que gestionan. Su interés se da mayormente en sistemas multiusuario, en los que resulta esencial mantener lo privado del trabajo de los usuarios, pero incluso en un sistema monousuario como el DOS, son de utilidad. En este sistema, cada archivo posee los siguientes *atributos*:

- `R` es el atributo de "sólo lectura". Cuando se activa (`+R`), no se puede borrar ni modificar el archivo que lo posee. La opción `-R` desactiva este atributo.
- La marca `A` indica que se desea hacer copia de seguridad, y es detectada por los programas `backup` y `xcopy`.

El programa `ATTRIB` nos permite establecer y conocer los atributos de un archivo o de un grupo de ellos:

[*posición*]`ATTRIB` [±`R`][±`A`] [*archivo(s)*]

Estos atributos se indican mediante letras:

`a`	archivo modificado
`r`	archivo de sólo lectura
`h`	archivo oculto: las órdenes `DIR` y `COPY` lo ignoran
`s`	archivo del sistema

A.4.3 Ampliaciones de la memoria en los PCs

Los primeros procesadores de IBM para computadores personales, tales como el 8088 y el 8086,[5] fueron diseñados para direccionar 1 Mb de memoria; sin embargo, los desarrollos posteriores del hardware y el software se encontraron pronto con limitaciones de memoria. Este problema obligó a introducir algunas ampliaciones del concepto de memoria. En este apartado se dará una visión sobre los diferentes tipos de memoria que se pueden encontrar en un PC trabajando bajo DOS.

Un PC puede tener tres tipos diferentes de memoria: *convencional, expandida* y *extendida*.

[5] Los computadores que usan estos procesadores se conocen como XT.

1. La *memoria convencional* es el primer megabyte de memoria del computador, con un procesador del tipo 8088 o superior. Éste era el único tipo de memoria disponible en los primeros PCs; un programa en ejecución se carga en los primeros 640 Kb de memoria convencional, y los restantes 384 Kb son utilizados por dispositivos de hardware.

2. La *memoria expandida* se añade al computador, y no forma parte de la memoria manejada directamente por el procesador. El acceso y la gestión de esta memoria se efectúa mediante un subsistema especial hardware/software.

3. La *memoria extendida* es la que está por encima del primer megabyte y sólo puede accederse mediante un procesador[6] 80286 ó superior.

A continuación se discute con mayor detalle la gestión de las memorias expandida o extendida.

Memoria expandida

La memoria expandida es el tipo de memoria más flexible que un PC puede tener, lo cual también hace que sea la más compleja de todas. Esta memoria permite al procesador acceder a más memoria de la disponible bajo el límite de los 640 Kb.

La memoria expandida no puede ser gestionada directamente por el DOS; para ello se utilizan programas (tales como EMM[7], EMM386 y QEMM) que se cargan normalmente en el archivo de configuración `config.sys`.

Es importante conocer cómo actúa el EMM para comprender mejor las diferencias entre las memorias expandida y extendida. Cuando el EMM se ejecuta lo que hace es buscar un trozo de memoria sin utilizar en los 384 Kb superiores del primer Mb; este trozo de memoria, usualmente de 64 Kb, se divide a su vez en cuatro *páginas de memoria* de 16 Kb que

[6]Los computadores que usan un procesador 80286 son conocidos como AT.
[7]Siglas del inglés *Expanded Memory Manager*.

hacen de pivotes para mover información entre la memoria convencional y la memoria expandida. El programa EMM se encarga de gestionar el intercambio de información entre las páginas de memoria y la memoria expandida.

La instalación de memoria expandida representa la instalación de un subsistema completo de hardware y software. El hardware consiste en una tarjeta de memoria diseñada especialmente, y el software es el archivo de gestión EMM.

La memoria expandida tiene características de una tarjeta de memoria normal y de un dispositivo periférico; tiene *chips* de memoria, pero se accede a ella como si se tratara de un puerto de entrada/salida. En la figura A.4 se intenta reflejar la forma en que el procesador ve a la memoria expandida.

Memoria extendida

La memoria extendida es la que está por encima del primer megabyte de memoria. Sólo puede accederse a ella a través del *modo protegido* de los procesadores 80286 o superior. La diferencia más importante entre memoria expandida y extendida es que los programas que trabajen en *modo real* no pueden tener acceso directo a la memoria extendida.

El modo protegido es usado para acceder a una mayor cantidad de memoria y de funciones.[8] Cuando un procesador (80286 o superior) pasa a modo protegido entonces el sistema puede controlar la operación de múltiples programas en la memoria y cambiar más fácilmente de una tarea a otra. Estos procesadores también pueden ejecutar los programas escritos para procesadores 8088 y 8086 trabajando en modo real, aunque en este caso no pueden acceder directamente a la memoria superior al primer Mb. Una de las principales ventajas de disponer de memoria extendida en un PC trabajando en modo real es su uso como un *disco virtual* en la memoria RAM (también llamado RAM-*drive*).

Dado que el acceso a la memoria es mucho más rápido que a un disco duro, el uso de un disco RAM disminuirá el tiempo que tarde

[8] Utilizando la mayor anchura del bus de direcciones.

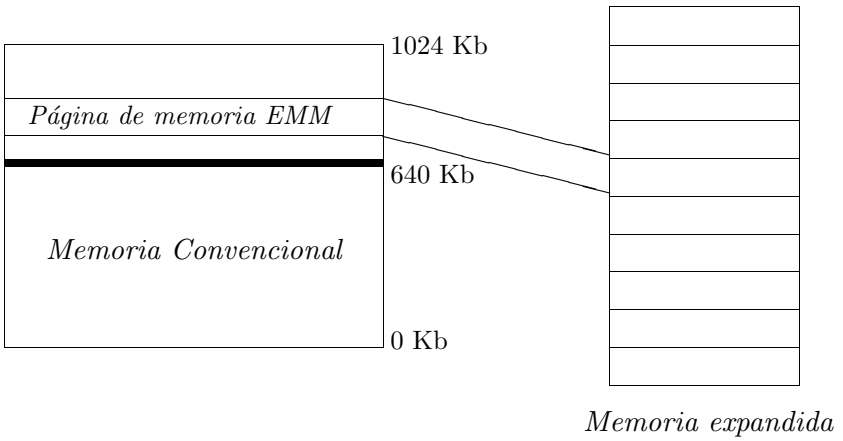


Figura A.4.

un programa en cargar los trozos de código necesarios para su correcto funcionamiento, pues la referencia a archivos contenidos en un disco RAM es la misma que si estuvieran en un disco duro. La desventaja del uso de un disco RAM es que, a diferencia de un disco duro o un disco flexible, la información escrita en un disco RAM es volátil (sólo permanece mientras el computador esté funcionando).

Apéndice B

Introducción a UNIX

El sistema operativo UNIX fue diseñado como un sistema de tiempo compartido con una interfaz de usuario (*shell*) simple y manejable, de la que existen distintas versiones.

Se trata de un sistema pensado inicialmente para equipos pequeños, por lo que los algoritmos fueron seleccionados por su simplicidad y no por su eficacia o rapidez.

Las ventajas que presenta UNIX sobre otros sistemas de gran difusión se basan en que el software desarrollado sobre UNIX es fácilmente transportable entre máquinas de distintos fabricantes, pues no necesita un hardware determinado y puede correr sobre procesadores de distinta filosofía.

No obstante lo anterior, los sistemas grandes hasta hace poco han preferido usar un sistema propietario a uno abierto, aunque la tendencia actual apunta hacia la adopción generalizada de este sistema por parte de todos los grandes fabricantes.

B.1 Breve descripción técnica

Considerando el sistema operativo como la capa de software situada directamente sobre el hardware, dentro de él podemos distinguir a su vez dos subcapas:

- La parte más próxima al hardware es lo que se denomina *núcleo* (en inglés *kernel*). En él es donde se encuentran los programas que controlan la gestión de archivos, la gestión de memoria y la planificación del uso de la UCP.

- La parte más próxima al usuario, un intermediario llamado *intérprete de comandos* o *shell*. El hecho de que sea el *shell* el intérprete entre el usuario y el sistema tiene ventajas bastante interesantes que se detallarán más adelante.

Una característica importante de UNIX es que *todo* se considera como un archivo (desde los ficheros de datos hasta los periféricos y las unidades de disco flexible). Un archivo para UNIX no es más que una secuencia de bytes sobre la que el sistema no impone estructura alguna, ni asigna significado a su contenido; el significado de los bytes depende únicamente de los programas que interpretan el archivo. Esta filosofía permite que hasta los distintos dispositivos hardware tengan un sitio dentro del sistema de archivos.

El sistema de archivos está organizado en estructura de árbol[1], y permite nombrar archivos usando rutas absolutas, que parten del directorio raíz, y rutas relativas, que son las que parten del directorio actual.

Es precisamente en la gestión de procesos donde mejor se puede apreciar la potencia de UNIX, pues aparecen los mecanismos que permiten que varios procesos trabajen sobre la máquina simultáneamente optimizando así el uso de la UCP.

UNIX emplea un sistema muy simple para crear y manipular procesos. Éstos se representan mediante bloques de control, y la información que existe en ellos se utiliza para la planificación de la UCP.

El planificador de la UCP está diseñado para facilitar el procesamiento multitarea asignando fracciones de tiempo a los procesos mediante el algoritmo *round-robin*: cada proceso tiene una prioridad asociada, de forma que cuanto más alto es el número asignado, menor es

[1] DOS tomó esta característica del sistema UNIX.

la prioridad, y cuanto más tiempo de la UCP ha consumido un proceso, menor se hace su prioridad.

No todos los sistemas UNIX utilizan el mismo mecanismo para la gestión de la memoria. Los primeros sistemas utilizaban exclusivamente el mecanismo de *swapping*: un proceso se retira tanto más fácilmente cuanto más tiempo lleve ocioso o haya estado en la memoria principal.

Otros sistemas utilizan la *paginación* (véase el apartado 4.2.6) para eliminar la fragmentación externa, existiendo sistemas que utilizan el procedimiento de *paginación por demanda* (en inglés *demand-paging*).

B.2 Una sesión con UNIX

Puesto que UNIX es un sistema multiusuario, lo que debemos hacer antes de iniciar una sesión de trabajo es presentarnos al sistema. Al arrancar el computador, o al efectuar una conexión remota, UNIX pedirá nuestro nombre de usuario del siguiente modo:

```
login:
```

el usuario debe responder con el nombre de usuario que le ha sido asignado por el administrador del sistema. Tras introducir el nombre de usuario, supongamos que es `popeye`, el sistema pedirá nuestra clave de acceso:

```
login:popeye
password:
```

Como es lógico, el sistema no nos permitirá acceder a menos que se introduzcan un nombre de usuario y su clave de acceso correctos (debe tenerse en cuenta que UNIX distingue entre letras mayúsculas y minúsculas) que el administrador debe habernos proporcionado. En la primera conexión al sistema, lo primero que debemos hacer es cambiar la clave de acceso de modo que nadie más la conozca (esto se hace mediante el comando `passwd`).

Si todo ha ido bien, el sistema aceptará la petición de entrada, nos colocará en nuestro directorio de trabajo y en pantalla aparecerá el *prompt* del sistema, que en el shell `sh` generalmente es `$`.

Un servicio importante que ofrece todo sistema UNIX es el de ayuda interactiva, llamada también ayuda en línea[2]. El comando `man` hace una llamada al *manual*, y puede ser usado para obtener información acerca de un comando, su sintaxis y sus posibles opciones (*flags*), su semántica e incluso de sus *bugs* o situaciones en las que dicho comando no realiza su labor correctamente. Por ejemplo,

```
$man passwd
```

da completa información acerca del comando `passwd`, recuérdese que `$` es el *prompt* del sistema, y no hay que teclearlo.

En un sistema multiusuario, uno puede estar interesado por saber qué personas están trabajando simultáneamente en el sistema. Esto se puede saber, y se dispone de dos comandos para ello: `who` y `finger`: la orden `who` proporciona los nombres de usuario de todas aquellas personas que se encuentran conectadas al sistema, mientras que `finger` proporciona información más detallada acerca de ellas.

Una vez que sepamos quiénes están conectados al sistema, podemos establecer contacto directo con ellos mediante los comandos `write` o `talk`, e incluso evitar ser molestados con mensajes de otros usuarios desactivando nuestra capacidad para recibir mensajes mediante `mesg -n`.

Al terminar una sesión de trabajo se debe cerrar la conexión con el sistema mediante `exit` o `logout` para impedir que alguien no autorizado pueda acceder al sistema. Sobre el asunto de la seguridad de los archivos se tratará en la siguiente sección.

B.3 Gestión de archivos

Este apartado resultará bastante fácil a aquellos usuarios que estén familiarizados con DOS, ya que muchas de las características del sistema de archivos de UNIX fueron adoptadas por DOS.

[2]En inglés, *on line*.

B.3.1 Identificadores

El nombre de un archivo puede ser casi cualquier secuencia de caracteres, aunque dos nombres se consideran el mismo si coinciden sus primeros catorce caracteres. Está permitido usar cualquier carácter ASCII en el nombre de un archivo salvo "`/`", que sirve para indicar el camino (absoluto o relativo) del nombre de un archivo; sin embargo, es mejor no complicarse la vida y usar sólo caracteres alfanuméricos.

Del mismo modo que en DOS, se puede hacer uso de caracteres comodín para referirse a un conjunto de archivos cuyo nombre verifica cierto patrón; esta labor la realiza el *shell*, y se detallará más adelante. Por último, no se debe olvidar que UNIX distingue entre letras mayúsculas y minúsculas.

B.3.2 Tipos de archivos en UNIX

En UNIX podemos encontrar varios tipos de archivos: los archivos ordinarios, los directorios, los vínculos, los vínculos simbólicos y los archivos especiales. De los archivos ordinarios y de los directorios poco hay que añadir a lo dicho en 4.2.2, por lo cual sólo incidiremos en los vínculos, simbólicos o no, y los archivos especiales.

Vínculos.- Un vínculo (en inglés *link*) no es más que otro nombre para un archivo. Esto tiene interés, además del evidente ahorro de espacio de almacenamiento, en aquellas situaciones en las que más de una persona está trabajando sobre un mismo archivo, por ejemplo un capítulo de un libro, cada autor puede tener en su directorio de trabajo un archivo (llamado por ellos respectivamente `cap1.tex`, `introduc.tex`, `ConceptosGenerales.tex`, ...) que "apunte" al mismo archivo en el disco.

Vínculos simbólicos.- Este tipo de archivos sólo contiene el nombre de otro archivo, que es el que se utiliza cuando el sistema operativo trabaja sobre el vínculo simbólico. Una aplicación bastante útil de los vínculos simbólicos, que también aclarará su función, aparece cuando el administrador realiza una modificación importante en la estructura de directorios; hasta que todos los usuarios se adapten

a la nueva ordenación, los vínculos simbólicos guiarán por la nueva estructura de árbol a los usuarios que aún no estén al tanto de la actualización.

Archivos especiales.- Los archivos especiales representan *dispositivos físicos* tales como terminales, impresoras, unidades de disco externo, unidades de cinta magnética o lectores de discos compactos. Esta forma de manejar el hardware permite trabajar al margen de las particularidades de los dispositivos físicos.

B.3.3 Permisos asociados con un archivo

Cada archivo de UNIX tiene una serie de permisos asociados con él. Estos permisos otorgan la posibilidad de que el archivo pueda ser leído, modificado o ejecutado, tanto por el propietario del archivo como por otros usuarios.

Los permisos asociados con un archivo pueden revisarse con la opción `-l` del comando `ls`:, que da un listado de todos los archivos del directorio actual junto con información adicional, como se muestra a continuación:

```
$ls -l
-rw-r-----  12  popeye  79850  Sep 18  20:50  ./tema1
-rw-r-----  11  popeye  76414  Sep 28  12:01  ./tema2
drw-r-----   1  popeye   1024  Sep 1   19:32  ./fig
```

Los permisos asociados a cada archivo aparecen codificados en la forma de una lista de caracteres:

1. El primer carácter indica el tipo de archivo: el signo `-` indica que se trata de un archivo ordinario, mientras que `d` indica que se trata de un directorio, los vínculos tienen una `l` y los archivos especiales tienen una `b` o una `c`, según se trate de dispositivos de almacenamiento por bloques o por caracteres.

2. Los siguientes tres caracteres, `rw-`, representan los permisos del dueño del archivo: `r` indica permiso de lectura, `w` permiso de escritura y `-` indica que el archivo *no* es ejecutable; si lo fuera aparecería una `x` en esa posición.

3. Los tres caracteres siguientes indican los permisos de los usuarios del grupo del dueño del archivo, y los tres últimos indican los permisos para otros usuarios, respectivamente. En el caso de los archivos relacionados más arriba se observa que los componentes del grupo sólo tienen permiso de lectura, no de escritura; los otros usuarios no pueden acceder a estos archivos, ni siquiera para leerlos.

En relación con los permisos de archivos se introduce el comando `chmod`, que sirve para cambiar los permisos asociados con un archivo. La sintaxis de este comando es la siguiente

```
$chmod nmk archivo
```

donde `n`, `m` y `k` son números del 0 al 7, que indican los permisos del dueño, de su grupo y de otros, respectivamente. Cada dígito octal (pues varía entre 0 y 7) se determina sumando 4 para activar el permiso de lectura, sumando 2 para activar el permiso de escritura y sumando 1 para el permiso de ejecución; así pues, para asignar al archivo `prueba` los permisos `-rwxrw-r--` tendríamos que escribir

```
$chmod 764 prueba
```

Naturalmente, los permisos sólo puede cambiarlos el propietario del archivo (y el administrador del sistema). Así por ejemplo, si "prestásemos" alguno de nuestros archivos a otro usuario, éste no podría cambiar sus permisos. Para que pudiera hacerlo, se debería cambiar la propiedad del archivo; por ejemplo, si el usuario `popeye` quiere ceder la propiedad del archivo `espinacas` al usuario `cocoliso` deberá hacer uso del comando `chown` (del inglés *CHange OWNer*):

```
$chown cocoliso espinacas
```

B.3.4 Órdenes para la gestión de archivos

Debido a que el uso principal de UNIX es manejar archivos, existen muchos comandos para manejarlos. Los nombres de los comandos suelen ser bastante similares a los de DOS y generalmente consisten en abreviaturas de su función. A continuación se presentan algunos de los

más comúnmente utilizados; la mayor parte de ellos tienen opciones, que pueden verse usando `man`:

- Los comandos más frecuentes para el manejo de directorios son:
  - `cd` (*Change Directory*) sirve para cambiar el directorio actual.
  - `pwd` (*Print Work Directory*), que proporciona el *path* del directorio actual.
  - `mkdir` (del inglés *MaKe DIRectory*), para crear un subdirectorio en el directorio actual.
  - `rmdir` (*ReMove DIRectory*), para borrar un subdirectorio del directorio actual.
- Los comandos principales para el manejo de archivos son:
  - `cat` para *conCATenar* archivos en pantalla, esto es, mostrar en la pantalla uno o varios archivos sucesivamente.
  - `cp` para *CoPiar* un archivo.
  - `find` para *encontrar* (en inglés *find*) un archivo dentro del árbol de directorios mediante su nombre o por alguna otra característica.
  - `ln` para establecer un *vínculo* (en inglés *LiNk*) o un *vínculo simbólico*.
  - `ls` para *LiStar* todos los archivos de un directorio.
  - `more` para listar el contenido de un archivo pantalla por pantalla.
  - `mv` para *trasladar* ("MoVer" en sentido físico) un archivo de un directorio a otro, o bien *renombrar* un archivo.
  - `rm` para *eliminar* (en inglés *ReMove*) un archivo del sistema de directorios.

B.4 El shell de UNIX

El *shell*, como intermediario entre el usuario y el núcleo del sistema, es el programa que durante más tiempo se estará usando durante una

sesión de trabajo con UNIX. En esta sección se hace una breve introducción a las características principales del (los) *shell* de UNIX.

Existen distintas versiones de *shell*, cada una de ellas con sus propias particularidades; en esta sección sólo se presentará sucintamente el *shell* `sh` del Sistema V. Otros *shell*, como el *shell C* (`csh`) y el *shell Korn* `ksh`, tienen las mismas características básicas que `sh` y proporcionan algunas utilidades adicionales.

Cada vez que se lanza `sh` se ejecuta el archivo `.profile` (el nombre comienza con un punto) que hace las veces del archivo `autoexec.bat` de DOS. Este archivo se utiliza para la personalización del entorno de trabajo: la definición de la variable `PATH`, fijar el tipo de terminal, cambiar el *prompt* por defecto, ...

B.4.1 Encauzamiento de la entrada y salida

Una primera utilidad del *shell* es la posibilidad de redireccionamiento de entrada/salida. Los operadores de redirección son los mismos que los de DOS: la salida se gestiona con `>` o con `>>` y la entrada se indica con `<`. Asimismo, es posible la concatenación de programas, enviando la salida de uno a la entrada del siguiente mediante el símbolo `|`. La idea, que consiste en hacer que la salida de un comando sea la entrada de otro sin necesidad de utilizar archivos temporales, ya debe ser conocida por los usuarios de DOS.

B.4.2 Caracteres comodín

Otra importante utilidad que podemos obtener de un *shell* es el uso de caracteres comodín. El *shell* permite el uso de caracteres comodín de tres tipos: `*`, `?` y `[...]`.

El signo de interrogación identifica a cualquier carácter, del mismo modo que en DOS; sin embargo, el asterisco es interpretado de forma distinta, como muestra el siguiente ejemplo:

```
$ls cap*tex
```

que hace un listado de todos los archivos del directorio actual cuyo nombre comience por `cap` y termine por `tex`.[3] El núcleo no "ve" el asterisco, "`*`", ya que éste es interpretado por el *shell* enviando al núcleo los archivos que satisfacen el patrón especificado.

El último tipo de carácter comodín permite indicar un rango de caracteres para confrontar con un carácter del nombre de un archivo, por ejemplo la orden

```
$cat cap[1-4].tex > parte1.tex
```

concatena los archivos `cap1.tex`, `cap2.tex`, `cap3.tex` y `cap4.tex` poniéndolos en el archivo `parte1.tex`. Entre los corchetes se pueden especificar rangos tanto numéricos como alfabéticos.

B.4.3 Guiones de shell

Las características de un *shell* permiten considerarlo como un lenguaje de programación, en el sentido de que es posible agrupar comandos que realizan tareas sencillas, definiendo así tareas más complejas. Esto se consigue mediante los llamados *guiones de shell* (en inglés *shell scripts*), que son la contrapartida UNIX de los archivos de extensión `.bat` de DOS.

B.5 UNIX como sistema multitarea

Como sistema multitarea UNIX proporciona herramientas para poder ejecutar varios programas simultáneamente; el *shell* permite, de forma fácil, controlar la ejecución de los distintos procesos mediante la introducción del concepto de ejecución en *modo subordinado* (en inglés *background*).

Cuando se ha de ejecutar un comando que va a durar mucho tiempo, como por ejemplo la búsqueda de un archivo determinado por toda la estructura de directorios, es conveniente ponerlo en modo subordinado, de modo que no haya que esperar a que termine para poder seguir trabajando con el *shell*. El símbolo `&` al final de una línea de comandos ejecuta los procesos en modo subordinado. Por ejemplo la línea de comandos

[3] Nótese la diferencia con DOS.

```
$find .  -name perdido -print> hallado &

[1] 1326
$
```

se ejecuta en modo subordinado buscando recursivamente en el directorio actual y sus subdirectorios (esto está indicado por el punto) el archivo `perdido` y, si lo encuentra, imprime su *path* absoluto en el archivo `hallado`.

Una vez que se ha creado un proceso en modo subordinado, el sistema imprime dos números y presenta el *prompt* a la espera de nuevas órdenes. El número entre corchetes es el *identificador del trabajo* y el otro número es el *identificador del proceso*.

El comando `jobs` visualiza todos los trabajos, junto con su identificador, que se están ejecutando en el *shell* actual. El identificador de trabajo se usa como argumento de comandos que permiten terminar un trabajo subordinado (`kill`); suspenderlo sin terminar (`stop`); reanudar en modo subordinado un trabajo suspendido (`bg`); y pasar un trabajo de modo subordinado a modo principal (`fg`). Por ejemplo

```
$stop %1
```

suspende el trabajo con el identificador `[1]`, y

```
$fg %1
```

reanuda el trabajo en modo principal. (Nótese que se usa el signo `%` antes de introducir el identificador de trabajo.)

Para la gestión eficiente del sistema, podemos modificar la prioridad con la que se ejecutan los procesos haciendo uso de los siguientes comandos:

`at` *hora orden* indica al sistema a qué hora se deberá ejecutar la *orden*.

`nice` *orden* ejecuta la orden recibida con una prioridad menor de lo normal. Esto es útil para ejecutar procesos que consumen gran cantidad de recursos y no corren prisa.

`nohup` *orden* ejecuta la orden recibida aunque el usuario se desconecte del sistema durante la ejecución. Útil para ejecutar procesos que consumen gran cantidad de tiempo.

B.6 Conclusión

Con todo lo anterior, lo más importante es comprender que, en general, cuando se emplea la palabra UNIX no se hace una simple referencia al núcleo del sistema operativo ni siquiera en sentido amplio, sino al núcleo del sistema junto con programas de aplicaciones destinados a crear un entorno de uso general.

Lo más importante de esta riqueza adicional puede resumirse en la posibilidad de que varios usuarios usen el computador al mismo tiempo (cosa que no puede hacerse en un PC bajo DOS) y en la posibilidad de que un usuario haga varias cosas a la vez señalando las prioridades que se desea; esto es, se trata de un sistema multitarea.

B.7 Prontuario de comandos UNIX

- Gestión de archivos y directorios.

  `cat` *archivo*
  : Concatena y muestra en la pantalla los archivos indicados.

  `cd` *path*
  : Cambia el directorio actual.

  `chmod` *nmk archivo*
  : Cambia los permisos (*nmk*) del archivo.

  `chown` *usuario archivo*
  : Cede a *usuario* la propiedad del archivo.

  `compress` *archivo*
  : Comprime el archivo.

`cp` (*archivo archivo* | *archivo directorio*)
: Realiza una copia del primer archivo bajo el nombre del segundo o copia el archivo en el directorio especificado.

`ln` *archivo archivo*
: Crea un vínculo entre los archivos dados.

`lp` *archivo*
: Imprime el archivo en papel.

`ls` *directorio*
: Lista el contenido de un directorio.

`mkdir` *directorio*
: Crea un subdirectorio en el directorio actual.

`more` *archivo*
: Imprime el archivo especificado pantalla por pantalla.

`mv` (*archivo archivo* | *archivo directorio*)
: Renombra el primer archivo con el nombre del segundo o mueve un archivo a un directorio.

`pwd`
: Muestra el nombre del directorio actual.

`rm` *archivo*
: Elimina un archivo.

`rmdir` *directorio*
: Elimina un directorio.

`uncompress` *archivo*
: Descomprime el archivo.

`zcat` *archivo*
: Muestra en pantalla la version descomprimida del archivo dado manteniéndolo comprimido.

- Comunicación con otros usuarios.

`mail`
: Gestión del correo electrónico.

`mesg` [*-y* | *-n*]
: Activa o desactiva la recepción de mensajes.

`talk` *usuario*
: Establece una conexión con otro usuario para intercambiar mensajes entre terminales.

`wall` *mensaje*
: Envía ese mensaje a todos los usuarios del sistema.

`write` *usuario*
: Establece una conexión con otro usuario para intercambiar mensajes.

- Utilidades.

`cal` *[mes][año]*
: Imprime la hoja del calendario del mes y año indicado.

`date` [*fecha y hora*]
: Imprime la fecha y hora actuales o especifica la nueva hora y fecha.

`finger`
: Muestra información detallada acerca de los usuarios conectados al sistema.

`who`
: Muestra los nombres de presentación de los usuarios conectados al sistema.

- Ayuda.

`man` *comando*
: Imprime en la pantalla la página del manual relativa al comando dado como argumento.

- Gestión de información.

`awk`
: Orden para la búsqueda y procesado de patrones.

`cmp` *archivo archivo*
: Compara los archivos y muestra la primera diferencia.

`comm` [*opciones*] *archivo archivo*
: Muestra líneas comunes o diferentes de los archivos dados.

`diff` *archivo archivo*
: Muestra los cambios necesarios para igualar los dos archivos dados.

`echo` *cadena*
: Escribe sobre la pantalla la cadena especificada.

`find` [*opciones*] [*característica de archivo*]
: Encuentra un archivo a partir de una de sus características dentro de un camino especificado.

`grep,egrep,fgrep` *expresión archivo*
: Busca apariciones de una expresión dentro de un archivo.

`sort` [*opciones*] *archivos*
: Ordena el contenido de uno o varios archivos, línea por línea.

`tail` *archivo*
: Visualiza en pantalla el final de un archivo.

`uniq` *entrada salida*
: Filtra las líneas repetidas del archivo de entrada y las escribe en el de salida.

`wc` *archivo*
: Cuenta líneas, palabras y caracteres de un archivo.

- Gestión de trabajos y procesos.

`at` *hora orden*
: Indica al sistema la hora en la que se deberán procesar las siguientes órdenes.

`bg` *identificador*
: Pasa el trabajo indicado a modo subordinado. Su nombre viene del inglés *background*.

`fg` *identificador*
: Pasa el trabajo indicado a modo principal. Su nombre viene del inglés *foreground*.

`jobs`
: Muestra el estado de los trabajos subordinados.

`kill` *identificador*
: Termina el trabajo indicado.

`nice` *comando*
: Ejecuta un comando con prioridad menor de la normal.

`nohup` *comando*
: Ejecuta un comando, incluso si después se suspende la sesión.

`ps`
: Muestra el estado de los procesos actuales.

`stop` *identificador*
: Suspende el trabajo indicado.

`tee` *archivo*
: Copia la entrada estándar a la salida estándar, además de al archivo.

`time` *comando*
: Informa acerca del uso de la UCP durante la ejecución de un comando.

- Procesamiento de textos.

`ed`
: Editor de archivos.

`nroff, troff`
: Procesadores de formateo de texto.

`sed`
: Editor en flujo.

`vi`
: Editor de pantalla.

Función	**Comando** DOS	**Comando** UNIX
Muestra la fecha	`DATE`	`date`
Muestra la hora	`TIME`	`date`
Muestra el contenido de un directorio	`DIR`	`ls`
Muestra el directorio actual	`CD`	`pwd`
Cambia el directorio actual	`CD` *path*	`cd` *path*
Crea un nuevo directorio	`MD, MKDIR` *dir*	`mkdir` *dir*
Suprime un directorio	`RD, RMDIR` *dir*	`rmdir, rm -r` *dir*
Muestra un archivo página a página	`MORE`	`more`
Copia un archivo	`COPY`	`cp`
Elimina un archivo	`DEL`	`rm`
Compara dos archivos	`COMP, FC`	`diff, cmp`
Renombrar un archivo	`REN(AME)`	`mv`
Mover un archivo		`mv`

Figura B.1. Comandos básicos de DOS y UNIX.

B.8 Diferencias entre DOS y UNIX

Aunque DOS está diseñado para sistemas PC monousuario, muchas de sus características están fuertemente influidas por el sistema UNIX. En particular, DOS se parece a UNIX en el diseño de su sistema de archivos, intérprete de comandos y en algunas de las órdenes de manipulación de archivos. Debido a esto, un usuario familiarizado con las órdenes de DOS no tendrá demasiados problemas para comenzar a manejar un sistema UNIX, o viceversa.

La mayor parte de los comandos usuales de DOS tienen su contrapartida UNIX, e incluso tienen nombres muy similares, por ejemplo `mkdir` está presente en ambos sistemas; la tabla de la figura B.1 muestra la equivalencia entre la mayoría de los comandos más usuales de DOS y UNIX.

Dejando a un lado las características multitarea y multiusuario de

UNIX, las diferencias más importantes entre DOS y UNIX para la línea de comandos y la gestión de archivos se enumeran a continuación:

1. *Distinción entre mayúsculas y minúsculas*: UNIX es sensible a las diferencias entre letras mayúsculas y minúsculas, mientras que DOS no lo es.

2. *Diagonal y diagonal inversa*: DOS usa la diagonal inversa `\` para los *paths*, por ejemplo en `\juegos\tetris`, mientras que UNIX usa la diagonal `/`, el *path* anterior se escribiría `/juegos/tetris`.

3. *Nombres de archivos*: en DOS los nombres de archivos están limitados a ocho caracteres alfanuméricos, seguidos opcionalmente de un punto y una extensión de tres letras; en UNIX los nombres de archivos pueden contener hasta catorce caracteres, pueden incluir uno o más puntos que no se tratan especialmente salvo cuando es el primer carácter del nombre.

4. *Símbolos comodín*: ambos sistemas permiten el uso de `*` o `?` para especificar conjuntos de nombres de archivos, aunque DOS es algo más rígido con `*`, que sólo se interpreta al final del nombre o la extensión de una plantilla. UNIX dispone además de una notación especial para indicar un rango de caracteres: así pues, `cap[1-3]` indicará los archivos `cap1`, `cap2` y `cap3`.

Bibliografía

[ACM91] ACM/IEEE. *Computing curricula*. Communications of the ACM, 34(6):69–84, 1991.

[Bai90] R. Bailey. *Functional Programming with Hope*. Ellis Horwood Ltd, 1990.

[Ber86] J. Bernstein. *La máquina analítica*. Editorial Labor. Barcelona, 1986.

[Bis91] P. Bishop. *Conceptos de Informática*. Anaya, 1991.

[BW88] R. Bird y P. Wadler. *Introduction to Functional Programming*. Prentice Hall International (UK) Ltd, 1988.

[Bye90] R. A. Byers. *Introducción a las bases de datos con dBASEIII plus*. McGraw-Hill, 1990.

[CM87] W. F. Clocksin y C. S. Mellish. *Programación en Prolog*. Gustavo Gili, S. A., 1987.

[Com87] *Computación avanzada y perspectivas de futuro*. Prensa Científica, S.A. Barcelona, Dic. 1987. Número especial de *Investigación y Ciencia*, dedicado a este tema.

[Dat93] C. J. Date. *An Introduction to Database System*. Volume 1, Addison-Wesley, 1993.

[DCG*89] P. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. B. Tucker, A. J. Turner y P.R. Young. *Computing as a discipline*. Communications of the ACM, 32(1):9–23, 1989.

[DDH72] O. J. Dahl, E. W. Dijkstra y C. A. R. Hoare. *Structured Programming*. Academic Press Ltd., 1972.

[Dei93] H. M. Deitel. *Sistemas operativos*. Addison Wesley iberoamericana, 1993.

[Dew88] A. K. Dewdney. *De la creación y ruptura de claves: primera parte*. Investigación y Ciencia, 147:136–141, 1988.

[Dew89] A. K. Dewdney. *Creación y ruptura de claves: segunda parte*. Investigación y Ciencia, 148:95–99, 1989.

[FM87] G. C. Fox y P. C. Messina. *Arquitecturas avanzadas de computadores*. Investigación y Ciencia, 135:24–33, 1987.

[For70] G. E. Forsyte. *Pitfalls in Computation, or why a math book isn't enough*. Technical Report, Computer Science Department, Stanford University, 1970.

[FSV87] G. Fernández y F. Sáez Vacas. *Fundamentos de Informátia*. Alianza Editorial. Madrid, 1987.

[GGSV93] J. Galve, J. C. González, A. Sánchez y J. A. Velázquez. *Algorítmica. Diseño y análisis de algoritmos funcionales e imperativos*. ra-ma, 1993.

[GL86] L. Goldschlager y A. Lister. *Introducción moderna a la Ciencia de la Computación con un enfoque algorítmico*. Prentice-Hall hispanoamericana. S.A. Méjico, 1986.

[Gol91] D. Goldberg. *What every computer scientist should know about floating-point arithmetic*. ACM Computing Surveys, 23(1):5–48, 1991.

[Gro86] P. Grogono. *Programación en Pascal*. Addison Wesley Iberoamericana, 1986.

[HH89] C. J. Hursch y J. L. Hursch. *SQL. El lenguaje de consulta estructurado*. ra-ma, 1989.

[Jam90] K. Jamsa. *DOS. Manual de referencia.* McGraw-Hill, 1990.

[JM90] B. Jacomy y J. Marguin. *De la machine à calculer de Pascal à l' ordinateur.* Musèe National des Techniques, CNAM. Paris, 1990.

[KM86] U. W. Kulisch y W. L. Miranker. *The arithmetic of the digital computer: a new approach.* SIAM Review, 24(1):671–677, 1986.

[Knu72] D. E. Knuth. *Ancient babylonian algorithms.* Communications of the ACM, 15(7), 1972.

[KP87] B.W. Kernighan y R. Pike. *El entorno de programación* UNIX. Prentice Hall Hispanoamericana, 1987.

[KS93] H. F. Korth y A. Silberschatz. *Fundamentos de bases de datos.* McGraw-Hill, 1993.

[La 91] *La revolución informática.* Prensa Científica, S.A. Barcelona, Nov. 1991. Número especial de *Investigación y Ciencia*, dedicado a la teleinformática.

[Lis86] A. M. Lister. *Fundamentos de los sistemas operativos.* Ed. Gustavo Gili, 1986.

[MA85] B. Mendizábal Allende. *Diccionario Oxford de Informática.* Díaz de Santos, S.A., 1985.

[Mac88] A. R. Mackintosh. *El computador del Dr. Atanasoff.* Investigación y Ciencia, 145:86–93, 1988.

[Mei73] J. P. Meinadier. *Estructura y funcionamiento de los computadores digitales.* Editorial AC, 1973.

[Mil89] M. Milenković. *Sistemas operativos: conceptos y diseño.* McGraw-Hill, 1989.

[MP93] A. de Miguel y M. Piattini. *Concepción y diseño de bases de datos: del modelo E/R al modelo relacional.* ra-ma, 1993.

[MW83] A. Mayne y M. Wood. *Introducción a las bases de datos relacionales.* Díaz de Santos, 1983.

[MW84] C. L. Morgan y M. Waite. *Introducción al microprocesador 8086/8088 (16 bits).* McGraw-Hill, 1984.

[Pas86] G. A. Pascoe. *Elements of object-oriented programming.* BYTE, 11(8):139–144, 1986.

[PLT89] A. Prieto, A. Lloris y J. Torres. *Introducción a la Informática.* McGraw-Hill, 1989.

[PS91] J. L. Peterson y A. Silberschapz. *Sistemas operativos: conceptos fundamentales.* Ed. Reverté, 1991.

[RRF91] K.H. Rosen, R.R. Rosinski y J.M. Farber. *UNIX Sistema V versión 4.* Mc Graw Hill, 1991.

[Rum83] S. M. Rump. *How reliable are results of computers?* Technical Report, Jarbuch berblicke Mathematik, 1983. Bibliographisches. Institut Mannheim, 1983.

[SH90] C. M. Stone y D. Hentchel. *Database wars revisited.* BYTE, 15(10):233–244, 1990.

[Sim85] G. L. Simmons. *Los ordenadores de la quinta generación.* Díaz de Santos. Madrid, 1985.

[SS86] L. Sterling y E. Shapiro. *The art of Prolog.* MIT Press, 1986.

[Stu75] *Study Group on Data Base Management Systems. Intern report.* ACM, 1975.

[Swa93] D. D. Swade. *La computadora mecánica de Charles Babbage.* Investigación y Ciencia, 199:66–71, 1993.

[Tes84] L. G. Tesler. *Lenguajes de programación.* Investigación y Ciencia, 98:36–45, 1984.

[Tho89] D. Thomas. *What's an object?* BYTE, 14(3):231–240, 1989.

[Tur92] *Turbo Pascal User Guide*. Borland International Inc., 1992.

[VJ85] A. Vaquero y L. Joyanes. *Informática. Glosario de términos y siglas*. McGraw-Hill, 1985.

[Weg89] P. Wegner. *Learning the language*. BYTE, 14(3):245–253, 1989.

[Wir86a] N. Wirth. *Algoritmos + Estructuras de datos = Programas*. Ediciones del Castillo. Madrid, 1986.

[Wir86b] N. Wirth. *Introducción a la programación sistemática*. El Ateneo, 1986.

Índice alfabético