

que los programas eran parte del procesador. El resultado de esta forma de pensar constituye un ejemplo palmario de situación en la que los árboles no dejan ver el bosque. Es fácil verse atrapado en este tipo de prejuicios erróneos y puede que el desarrollo de las Ciencias de la computación tenga todavía muchos de estos prejuicios sin que ni siquiera seamos conscientes de ellos. De hecho, parte del interés que la ciencia despierta se debe a que los nuevos conceptos están constantemente abriendo puertas hacia nuevas teorías y aplicaciones.

Cuestiones y ejercicios

1. ¿Qué secuencia de sucesos cree que harían falta para desplazar el contenido de una celda de memoria de una computadora a otra celda de memoria?
2. ¿Qué información debe suministrar el procesador a los circuitos de la memoria principal para escribir un valor en una celda de memoria?
3. Tanto el almacenamiento masivo como la memoria principal y los registros de uso general son sistemas de almacenamiento. ¿Qué diferencia hay en el uso que se da a estos distintos sistemas?

2.2 Lenguaje máquina

Con el fin de aplicar el concepto de programa almacenado, los procesadores están diseñados para reconocer instrucciones codificadas como patrones de bits. Este conjunto de instrucciones junto con el sistema de codificación utilizado forman lo que se conoce como **lenguaje máquina**. Una instrucción expresada en este lenguaje se denomina **instrucción de nivel máquina**.

Repertorio de instrucciones

La lista de instrucciones de lenguaje máquina que un procesador típico es capaz de decodificar y ejecutar es bastante corta. De hecho, una vez que una máquina puede realizar ciertas tareas elementales pero convenientemente elegidas, el añadir más funcionalidad no incrementa las capacidades teóricas de la máquina. En otras palabras, más allá de un cierto punto, la funcionalidad adicional puede proporcionar algo más de comodidad, pero no añade nada a las capacidades fundamentales de la máquina.

El grado con el que el diseño de un máquina debe aprovechar este hecho ha conducido a dos filosofías distintas de arquitecturas de procesador. Una de ellas es que un procesador debe diseñarse para ejecutar un conjunto mínimo de instrucciones en lenguaje máquina. Esta técnica conduce a lo que se denomina arquitectura **RISC** (*Reduced Instruction Set Computer*, Computadora de conjunto reducido de instrucciones). El argumento en favor de la arquitectura RISC es que las máquinas de ese tiempo son eficientes, rápidas y más baratas

¿Quién inventó el qué?

Asignar a una sola persona todo el mérito de un invento suele ser bastante dudoso. A Thomas Edison se le reconoce la invención de la lámpara incandescente, pero otros investigadores estaban desarrollando lámparas similares y podríamos considerar, en un cierto sentido, que Edison tuvo suerte en ser el que obtuvo la patente. A los hermanos Wright se les reconoce la invención del aeroplano, pero competían con otros muchos personajes contemporáneos y se beneficiaron del trabajo realizado por ellos, todos los cuales les deben algo, a su vez, a Leonardo da Vinci, que jugueteó con la idea de máquinas voladoras ya en el siglo xv. Incluso los diseños de Leonardo estaban aparentemente basados en ideas anteriores. Por supuesto, en estos casos el inventor reconocido sigue pudiendo reclamar legítimamente el mérito que le corresponde. En otros casos, la historia parece haber concedido ese mérito de manera inapropiada, un ejemplo sería el concepto de programa almacenado. Sin ninguna duda, John von Neumann era un científico brillante que se merece un reconocimiento por sus numerosas contribuciones, pero una de las contribuciones por las que la historia popular ha elegido concederle el mérito, el concepto de programa almacenado, fue aparentemente desarrollado por un equipo de investigadores dirigido por J. P. Eckert en la Escuela Moore de Ingeniería Eléctrica de la universidad de Pensilvania. John von Neumann fue simplemente el primero en publicar un trabajo en el que informaba acerca de dicha idea, lo cual es la razón de que en el campo de la mitología informática se le haya seleccionado como el inventor de ese concepto.

de fabricar. Por el contrario, otros diseñadores argumentan en favor de procesadores que tengan la capacidad de ejecutar un gran número de instrucciones complejas, aún cuando muchas de ellas sean técnicamente redundantes. El resultado de este enfoque se conoce con el nombre de **CISC** (*Complex Instruction Set Computer*, Computadora de conjunto complejo de instrucciones). El argumento en favor de la arquitectura CISC es que los procesadores más complejos pueden enfrentarse mejor a la complejidad cada vez mayor del software actual. Con una arquitectura CISC, los programas pueden aprovecharse de la existencia de un conjunto rico y potente de instrucciones, muchas de las cuales requerirían una secuencia multi-instrucción en un diseño RISC.

En la década de 1990 y en la primera década de este milenio, los procesadores CISC y RISC comercialmente disponibles han estado compitiendo activamente por el papel predominante en el campo de los equipos de sobremesa. Los procesadores de Intel, utilizados en los PC, son un ejemplo de arquitectura CISC; los procesadores PowerPC (desarrollados mediante una alianza entre Apple, IBM y Motorola) son ejemplos de arquitectura RISC y fueron utilizados en el Apple Macintosh. A medida que ha ido pasando el tiempo, el coste de fabricación de los procesadores CISC se ha reducido enormemente; por ello, los procesadores de Intel (o sus equivalentes de AMD, Advanced Micro Devices, Inc.) pueden ahora encontrarse en todas las computadoras de sobremesa y portátiles (incluso Apple está ahora construyendo computadoras basadas en los productos de Intel).

Aunque la arquitectura CISC se ha garantizado un puesto predominante en las computadoras de sobremesa, tiene un insaciable apetito de potencia eléc-

trica. Por el contrario, la empresa Advanced RISC Machine (ARM) ha diseñado una arquitectura RISC específicamente pensada para un bajo consumo. (Advanced RISC Machine fue originalmente Acorn Computers y ahora se conoce como ARM Holdings.) Por tanto, los procesadores basados en el diseño de ARM y que son fabricados por diversas empresas, entre las que se incluyen Qualcomm y Texas Instruments, están presentes hoy día en controladoras de juegos, televisiones digitales, sistemas de navegación, módulos para automoción, teléfonos celulares, teléfonos inteligentes y otros dispositivos de electrónica de consumo.

Independientemente de la elección que se haga entre RISC y CISC, las instrucciones de una máquina pueden clasificarse en tres grupos: (1) el grupo de transferencia de datos, (2) el grupo aritmético/lógico y (3) el grupo de control.

Transferencia de datos El grupo de transferencia de datos está compuesto por instrucciones que solicitan el movimiento de datos desde una ubicación a otra. Los pasos 1, 2 y 4 de la Figura 2.2 caen dentro de esta categoría. Es preciso recalcar que el uso de términos tales como *transferir* o *mover* para identificar a este grupo de instrucciones es en realidad engañoso. Es raro que los datos que se están transfiriendo se borren de su ubicación original. El proceso implicado en una instrucción de transferencia es más una copia de los datos que un movimiento de los mismos. Por tanto, otros términos como *copiar* o *clonar* permitirían describir mejor las acciones llevadas a cabo por este grupo de instrucciones.

Sin salirnos del campo de la terminología, debemos mencionar que suelen emplearse términos especiales a la hora de hacer referencia a la transferencia de datos entre el procesador y la memoria principal. Una solicitud para llenar un registro de uso general con el contenido de una celda de memoria se suele denominar instrucción LOAD (instrucción de carga); a la inversa, una solicitud para transferir el contenido de un registro a una celda de memoria se denomina instrucción STORE (instrucción de almacenamiento). En la Figura 2.2, los pasos 1 y 2 especifican instrucciones LOAD y el Paso 4 indica una instrucción STORE.

Instrucciones de longitud variable

Para simplificar las explicaciones a lo largo del texto, el lenguaje máquina utilizado para los ejemplos de este capítulo (y descrito en el Apéndice C) utiliza un tamaño fijo (dos bytes) para todas las instrucciones. Por tanto, para cargar una instrucción, el procesador siempre extrae el contenido de dos celdas de memoria consecutivas e incrementa el contador de programa en dos unidades. Este comportamiento predecible simplifica la tarea de carga de las instrucciones y es característica de las máquinas RISC. Sin embargo, las máquinas CISC tienen lenguajes máquina con instrucciones de longitud variable. Los procesadores de Intel actuales, por ejemplo, tienen instrucciones que van desde las de un único byte hasta otras de múltiples bytes cuya longitud depende de la utilización exacta de dicha instrucción. Los procesadores con este tipo de lenguaje máquina determinan la longitud de instrucción que hay que cargar analizando el código de operación de dicha instrucción. Es decir, el procesador carga primero el código de operación de la instrucción y luego, dependiendo del patrón de bits recibido, sabe cuántos más bits debe cargar de la memoria para obtener el resto de la instrucción.

Un grupo importante de instrucciones dentro de la categoría de transferencia de datos está formado por los comandos utilizados para comunicarse con dispositivos externos al contexto definido por el procesador y la memoria principal (impresoras, teclados, pantallas, unidades de disco, etc.). Puesto que estas instrucciones se encargan de generar las actividades de entrada/salida, E/S (I/O, Input/Output), de la máquina se denominan **instrucciones de E/S** (o, en inglés, instrucciones I/O) y, en ocasiones, se las considera una categoría de instrucciones diferente. Por otro lado, en la Sección 2.5 se describe cómo pueden gestionarse estas actividades de E/S mediante el mismo conjunto de instrucciones que solicita la transferencia de datos entre el procesador y la memoria principal. Por tanto, vamos a considerar las instrucciones de E/S como parte del grupo de transferencia de datos.

Aritmético/Lógico El grupo aritmético/lógico está compuesto por aquellas instrucciones que le dicen a la unidad de control que debe solicitar una cierta actividad dentro de la unidad aritmético/lógica. El paso 3 de la Figura 2.2 cae dentro de este grupo. Como su propio nombre sugiere, la unidad aritmético/lógica puede realizar también otras operaciones diferentes de las operaciones aritméticas básicas. Algunas de estas operaciones adicionales básicas son las operaciones booleanas AND, OR y XOR, presentadas en el Capítulo 1 y que analizaremos con más detalle posteriormente.

Otro conjunto de operaciones disponible dentro de la mayor parte de las unidades aritmético/lógicas permite pasar el contenido de los registros hacia la derecha o hacia la izquierda sin salir del propio registro. Estas operaciones se conocen con el nombre de operaciones SHIFT (desplazamiento) o ROTATE (rotación), dependiendo de si los bits que se "caen" por el extremo del registro se descartan simplemente (SHIFT) o se utilizan para llenar el hueco que queda en el otro extremo (ROTATE).

Control El grupo de control está compuesto por aquellas instrucciones que dirigen la ejecución del programa en lugar de la manipulación de los datos. El paso 5 de la Figura 2.2 cae dentro de esta categoría, aunque se trata de un ejemplo

Figura 2.3 División de valores almacenados en la memoria.

- Paso 1. CARGAR (LOAD) un registro con un valor de la memoria.
- Paso 2. CARGAR (LOAD) otro registro con otro valor de la memoria.
- Paso 3. Si este segundo valor es cero, SALTAR (JUMP) al Paso 6.
- Paso 4. Dividir el contenido del primer registro entre el segundo registro y colocar el resultado en un tercer registro.
- Paso 5. ALMACENAR (STORE) el contenido del tercer registro en memoria.
- Paso 6. STOP.

bastante elemental. Este grupo contiene muchas de las instrucciones más interesantes del repertorio de una máquina, como la familia de instrucciones JUMP (o BRANCH), instrucciones de salto o bifurcación, que se utilizan para ordenar al procesador que ejecute una instrucción distinta de la que se encuentra a continuación en la lista. Existen dos tipos de instrucciones JUMP: **saltos incondicionales** y **saltos condicionales**. Un ejemplo de salto incondicional sería la instrucción “Saltar al paso 5”; mientras que un ejemplo de salto condicional sería: “Si el valor obtenido es 0, entonces saltar al paso 5.” La diferencia es que un salto condicional solo provoca el salto si se satisface una cierta condición. Por ejemplo, la secuencia de instrucciones de la Figura 2.3 representa un algoritmo para dividir dos valores, en el que el paso 3 es un salto condicional que nos protege frente a la posibilidad de división por cero.

Un ejemplo de lenguaje máquina

Vamos a ver ahora cómo se codifican las instrucciones de una computadora típica. La máquina que vamos a utilizar para nuestro análisis se describe en el Apéndice C y se ilustra en la Figura 2.4. Dispone de 16 registros de uso general y de 256 celdas en la memoria principal, cada una de ellas con una capacidad de 8 bits. Para propósitos de referencia, vamos a etiquetar los registros con los valores de 0 a 15 y las direcciones de las celdas de memoria con los valores 0 a 255. Por comodidad, vamos a considerar que estas etiquetas y direcciones son valores representados en base dos y vamos a expresar los patrones de bits resultantes en notación hexadecimal. Etiquetaremos los registros de 0 a F y las direcciones de las celdas de memoria de 00 a FF.

La versión codificada de una instrucción en lenguaje máquina está compuesta de dos partes: el campo de **código de operación** y el campo **operando**. El patrón de bits que aparece en el campo correspondiente al código de operación nos indica cuál es la operación elemental (como por ejemplo STORE, SHIFT, XOR o JUMP) solicitada por la instrucción. Los patrones de bits contenidos en el campo operando proporcionan información más detallada acerca de la opera-

Figura 2.4 Arquitectura de la máquina descrita en el Apéndice C.

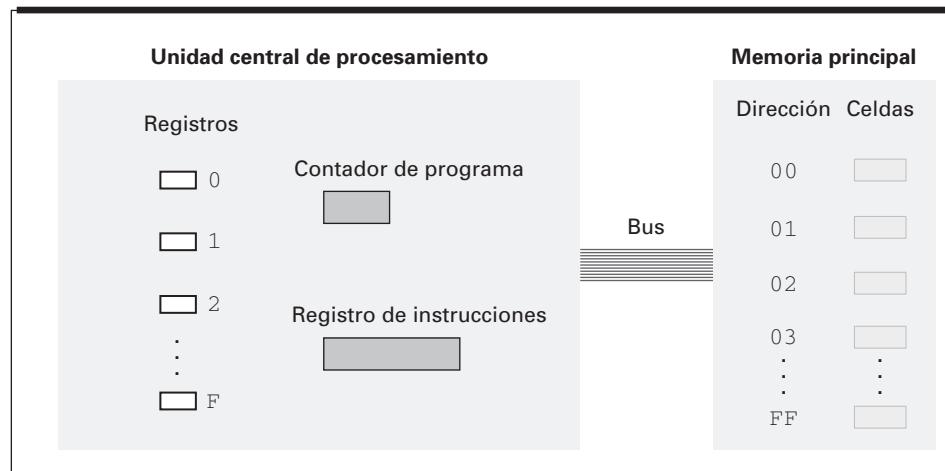
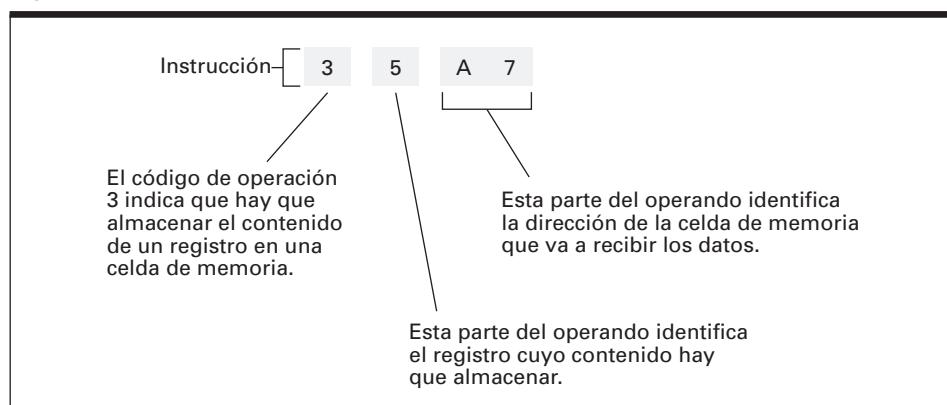


Figura 2.5 Composición de una instrucción para la máquina del Apéndice C.

ción especificada por el código de operación. Por ejemplo, en el caso de una operación STORE, la información del campo operando indica qué registro contiene el dato que hay que almacenar y qué celda de memoria tiene que recibir el dato.

El lenguaje máquina completo de nuestra máquina de ejemplo (Apéndice C) está compuesto por solo dos instrucciones básicas. Cada una de estas instrucciones se codifica utilizando un total de 16 bits, representados mediante cuatro dígitos hexadecimales (Figura 2.5). El código de operación de cada instrucción está compuesto por los primeros cuatro bits (o lo que es lo mismo, el primer dígito hexadecimal). Observe (Apéndice C) que estos códigos de operación están representados por los dígitos hexadecimales 1 a C. En particular, la tabla del Apéndice C muestra que una instrucción que comienza con el dígito hexadecimal 3 hace referencia a la instrucción STORE, mientras que una instrucción que comienza con el dígito hexadecimal A hace referencia a la instrucción ROTATE.

El campo de operando de cada instrucción de nuestra máquina de ejemplo está compuesto por tres dígitos hexadecimales (12 bits) y, en todos los casos, excepto para la instrucción HALT (que es la instrucción de detención y no necesita ningún detalle adicional), permite clarificar la instrucción general especificada por el código de operación. Por ejemplo (Figura 2.6), si el primer dígito hexadecimal de una instrucción fuera 3 (el código de operación para almacenar el contenido de un registro), el siguiente dígito hexadecimal de la instrucción nos indicaría qué registro es el que hay que almacenar y los dos

Figura 2.6 Decodificación de la instrucción 35A7.

últimos dígitos hexadecimales nos dirían en qué celda de memoria hay que almacenar ese dato. Por tanto, la instrucción 35A7 (hexadecimal) se traduce en la instrucción “Almacenar (STORE) el patrón de bits contenido en el registro 5, depositándolo en la celda de memoria cuya dirección es A7”. (Observe cómo simplifica las explicaciones el uso de la notación hexadecimal. En realidad, la instrucción 35A7 es el patrón de bits 0011010110100111.)

La instrucción 35A7 también proporciona un ejemplo explícito de por qué la capacidad de la memoria principal se mide en potencias de dos. Puesto que hemos reservado 8 bits de la instrucción para especificar la celda de memoria utilizada por la instrucción, es posible referenciar exactamente 2^8 celdas diferentes. Esto nos fuerza a construir una memoria principal que tenga exactamente este número de celdas, cuyas direcciones irán de 0 a 255. Si la memoria principal tuviera más celdas, no podríamos escribir instrucciones que distinguieran unas celdas de otras. Por el contrario, si la memoria principal tuviera menos celdas, existiría la posibilidad de escribir instrucciones que hicieran referencia a celdas inexistentes.

Veamos otro ejemplo de cómo se emplea el campo operando para clarificar la instrucción general dada por el código de operación: considere una instrucción con el código de operación 7 (hexadecimal), que solicita llevar a cabo la operación OR con el contenido de dos registros (ya veremos lo que significa combinar mediante OR dos registros en la Sección 2.4. Por ahora, lo que nos interesa simplemente es el modo en que las instrucciones se codifican). En este caso, el siguiente dígito hexadecimal indica el registro en el que hay que almacenar el resultado, mientras que los dos últimos dígitos hexadecimales indican cuáles son los dos registros que hay que combinar mediante OR. Por tanto, la instrucción 70C5 se traduce en la instrucción “combinar mediante OR el contenido del registro C y el contenido del registro 5 y almacenar el resultado en el registro 0”.

Existe una distinción sutil entre las dos instrucciones LOAD de nuestra máquina. Aquí podemos ver que el código de operación 1 (hexadecimal) identifica una instrucción que carga un registro con el contenido de una celda de memoria, mientras que el código de operación 2 (hexadecimal) identifica una instrucción que carga un registro con un cierto valor concreto. La diferencia es que el campo de operación en una instrucción del primer tipo contendrá una dirección, mientras que en el segundo tipo, el campo de operando contendrá el propio patrón de bits que hay que cargar.

Observe que la máquina dispone de dos instrucciones ADD: una para sumar representaciones en complemento a dos y otra para sumar representaciones en punto flotante. Esta distinción es una consecuencia del hecho de que la suma de patrones de bits que representan valores codificados en notación en complemento a dos requiere llevar a cabo, dentro de la unidad aritmético/lógica, una serie de actividades diferente de las que son necesarias para sumar valores en notación de punto flotante.

Vamos a cerrar esta sección examinando la Figura 2.7, que contiene una versión codificada de las instrucciones de la Figura 2.2. Hemos supuesto que los valores que hay sumar están almacenados en notación de complemento a dos en las direcciones de memoria 6C y 6D y que la suma hay que almacenarla en la celda de memoria situada en la dirección 6E.

Figura 2.7 Versión codificada de las instrucciones de la Figura 2.2.

Instrucciones codificadas	Traducción
156C	Cargar el registro 5 con el patrón de bits almacenado en la celda de memoria situada en la dirección 6C.
166D	Cargar el registro 6 con el patrón de bits almacenado en la celda de memoria situada en la dirección 6D.
5056	Sumar el contenido del registro 5 con el del registro 6 como si fueran representaciones en complemento a dos y almacenar el resultado en el registro 0.
306E	Almacenar el contenido del registro 0 en la celda de memoria situada en la dirección 6E.
C000	Parar.

Cuestiones y ejercicios

1. ¿Por qué el término *movimiento* podría considerarse incorrecto para la operación de mover datos de una ubicación a otra, dentro de la máquina?
2. En el texto, las instrucciones JUMP se expresaban identificando explícitamente el destino, indicando el nombre (o número de paso) de dicho destino dentro de la instrucción JUMP (por ejemplo, “Saltar al Paso 6”). Una desventaja de esta técnica es que si el nombre (o el número) se cambia posteriormente, nos vemos obligados a localizar todos los saltos a dicha instrucción y cambiar también el nombre. Describa otra forma de expresar una instrucción JUMP de modo que no haya que indicar explícitamente el nombre del destino.
3. ¿A qué categoría pertenece la instrucción “Si 0 es igual a 0, entonces saltar al Paso 7”, a la de los saltos condicionales o incondicionales? Explique su respuesta.
4. Escriba el programa de ejemplo de la Figura 2.7 utilizando patrones de bits reales.
5. Observe las siguientes instrucciones escritas en el lenguaje máquina descrito en el Apéndice C. Reescriba esas instrucciones en español normal.
 - a. 368A
 - b. BADE
 - c. 803C
 - d. 40F4
6. ¿Cuál es la diferencia entre las instrucciones 15AB y 25AB en el lenguaje máquina del Apéndice C?

nas gestionan las operaciones de E/S han provocado históricamente que el "mismo" lenguaje tenga diferentes características, o dialectos, en las distintas máquinas. En consecuencia, a menudo es necesario realizar modificaciones de carácter menor a un programa, antes de poder moverlo de una máquina a otra.

Para complicar aún más este problema de la portabilidad, existe una falta de acuerdo en algunos casos sobre cuál es la definición correcta de un lenguaje concreto. Para ayudar en este sentido, ANSI (American National Standards Institute) e ISO (International Organization for Standardization) han adoptado y publicado estándares para muchos de los lenguajes más populares. En otros casos, debido a la popularidad de un cierto dialecto de un lenguaje y al deseo de otros desarrolladores de compiladores de diseñar productos compatibles, se han desarrollado estándares informales. Sin embargo, incluso en el caso de los lenguajes altamente estandarizados, los diseñadores de compiladores a menudo proporcionan funcionalidades, que en ocasiones se denominan extensiones del lenguaje, que no forman parte de la versión estándar del mismo. Si un programador aprovecha estas funcionalidades, el programa generado no será compatible con aquellos entornos en los se emplee un compilador de un fabricante diferente.

En la historia de los lenguajes de programación, el hecho de que los lenguajes de tercera generación no llegaran a proporcionar una verdadera independencia con respecto a la máquina tiene, en la práctica, poca importancia por dos razones distintas. En primer lugar, estaban tan cerca de ser auténticamente independientes con respecto a la máquina que el software podía moverse de una máquina a otra con relativa facilidad. En segundo lugar, el objetivo de la independencia con respecto a la máquina resultó ser tan solo la semilla de otros objetivos mucho más ambiciosos. De hecho, el darse cuenta de que las máquinas podían responder a sentencias de alto nivel tales como

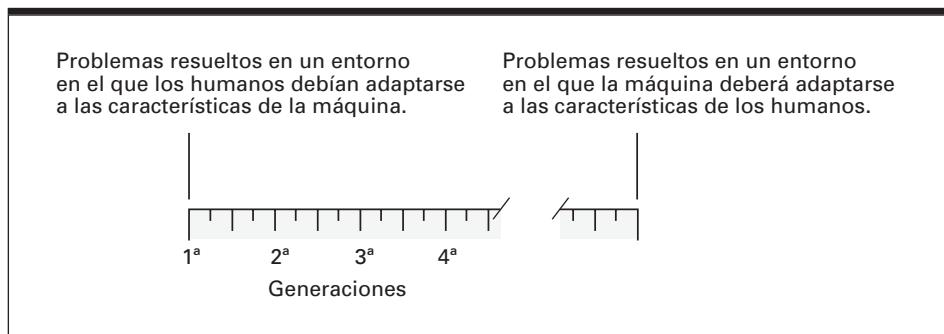
```
assign CosteTotal the value Precio + CosteTransporte
```

llevó a los expertos en computación a soñar con la creación de entornos de programación que permitieran a los humanos comunicarse con las máquinas en términos de conceptos abstractos en lugar de forzar a las personas a traducir esos conceptos a un formato compatible con la máquina. Además, esos expertos querían disponer de máquinas que pudieran realizar buena parte del proceso de descubrimiento de algoritmos, en lugar de limitarse a ejecutar esos algoritmos. El resultado ha sido un espectro cada vez más amplio de lenguajes de programación que desafía todos los intentos de clasificarlos de manera clara en términos de generaciones.

Paradigmas de programación

La clasificación en generaciones de los lenguajes de programación está basada en una escala lineal (Figura 6.1), en la que la posición de un lenguaje está determinada por el grado en que el usuario de ese lenguaje se ve liberado del mundo de las especificidades técnicas de las computadoras, pudiendo así pensar en términos meramente asociados con el problema que se pretende resolver. En la realidad, el desarrollo de los lenguajes de programación no ha progresado de esta manera, sino de que se ha desarrollado a lo largo de diferentes caminos a medida que han ido apareciendo una serie de enfoques alternativos del proceso

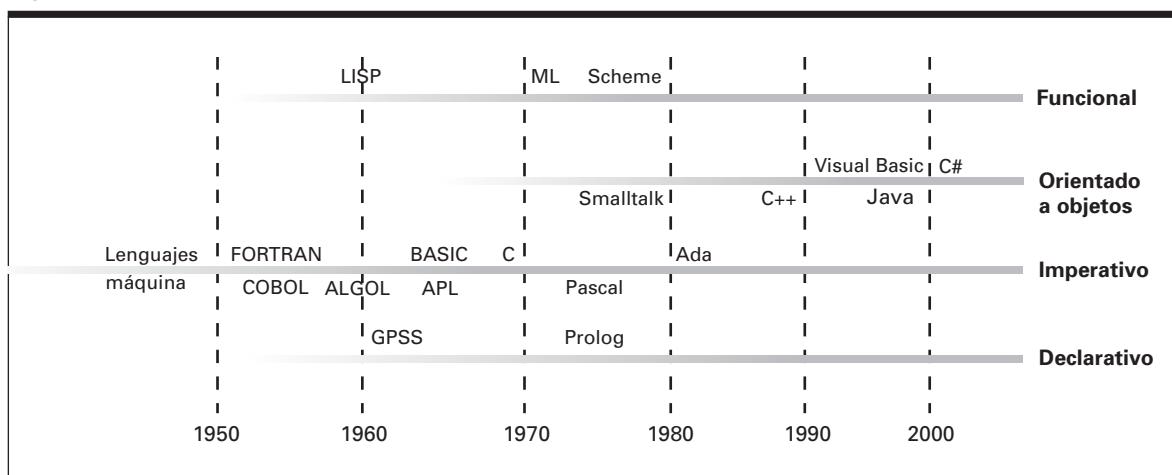
Figura 6.1 Generaciones de lenguajes de programación.



de programación (denominados **paradigmas de programación**). En consecuencia, podemos representar mejor el desarrollo histórico de los lenguajes de programación mediante un diagrama con múltiples rutas paralelas, como se muestra en la Figura 6.2, en la que se muestra cómo las diferentes rutas resultantes de los distintos paradigmas van emergiendo y progresando de manera independiente. En particular, la figura muestra cuatro rutas distintas, que representan los paradigmas funcional, orientado a objetos, imperativo y declarativo, estando colocados los distintos lenguajes asociados con cada uno de los paradigmas de forma tal que la figura indica su nacimiento en relación a otros lenguajes. (Esto no implica que un lenguaje haya evolucionado necesariamente a partir de otro anterior.)

Es preciso observar que aunque los paradigmas identificados en la Figura 6.2 se denominan *paradigmas de programación*, estas alternativas tienen ramificaciones que van más allá del proceso de programación propiamente dicho. Representan enfoques fundamentalmente distintos para obtener soluciones a los problemas y, por tanto, afectan al proceso completo de desarrollo software. En este sentido, el término *paradigma de programación* induce a confusión. Un término más realista sería *paradigma de desarrollo software*.

Figura 6.2 Evolución de los paradigmas de programación.



El **paradigma imperativo**, también conocido como **paradigma procedimental**, representa el enfoque tradicional del proceso de programación. Es el paradigma en el que está basado nuestro pseudocódigo del Capítulo 5, así como el lenguaje máquina visto en el Capítulo 2. Como su nombre sugiere, el paradigma imperativo define el proceso de programación como el desarrollo de una secuencia de comandos que, al ser ejecutados, manipula los datos para generar el resultado deseado. Por tanto, el paradigma imperativo nos dice que debemos enfocar el proceso de programación determinando un algoritmo para solucionar el problema que nos traemos entre manos y luego expresando dicho algoritmo como una secuencia de sentencias.

A diferencia del paradigma imperativo, el **paradigma declarativo** pide al programador que describa el problema que hay que resolver, en lugar del algoritmo que hay que aplicar. Para ser más precisos, un sistema de programación declarativo aplica un algoritmo preestablecido para resolución de problemas de propósito general con el fin de solucionar los problemas que se le presenten. En un entorno de este tipo, la tarea del programador consiste en desarrollar un enunciado preciso del problema en lugar de describir un algoritmo para la resolución del problema.

Uno de los principales obstáculos a la hora de desarrollar sistemas de programación basados en el paradigma declarativo se encuentra en la necesidad de disponer de un algoritmo subyacente para la resolución del problema. Por esta razón, los primeros lenguajes de programación declarativos tendían a ser de propósito especial por su propia naturaleza, habiendo sido diseñados para su uso en aplicaciones concretas. Por ejemplo, el enfoque declarativo ha sido utilizado durante muchos años para simular sistemas (políticos, económicos, medioambientales, etc.) con el fin de probar hipótesis o de realizar predicciones. En este tipo de entornos, el algoritmo subyacente es básicamente el proceso de simular el paso del tiempo recalculando de forma repetitiva los valores de una serie de parámetros (producto interior bruto, déficit comercial, etc.) a partir de los valores anteriormente calculados. Así, la implementación de un lenguaje declarativo para llevar a cabo simulaciones de este tipo requiere implementar primero un algoritmo que se encargue de llevar a la práctica ese procedimiento repetitivo. Entonces, la única tarea que un programador que use el sistema tiene que llevar a cabo es describir la situación que se desea simular. De esta manera, un meteorólogo no necesita desarrollar un algoritmo para predecir si lloverá o no, sino que simplemente tendrá que describir el estado meteorológico actual, permitiendo al algoritmo de simulación subyacente generar las predicciones del tiempo para los próximos días.

El paradigma declarativo sufrió un tremendo impulso cuando se descubrió que el tema de la lógica formal dentro del campo de las matemáticas proporciona un algoritmo de resolución de problemas simple que es adecuado para su utilización en un sistema de programación declarativa de propósito general. El resultado ha sido que ahora se presta mucha más atención al paradigma declarativo y que ha surgido la denominada **programación lógica**, un tema que veremos en la Sección 6.7.

Otro paradigma de programación es el **paradigma funcional**. En este caso, un programa se ve como una entidad que acepta entradas y genera salidas. Los matemáticos denominan a tales entidades funciones, razón por la que esta técnica recibe el nombre de paradigma funcional. Bajo este paradigma, los

programas se construyen conectando entidades predefinidas más pequeñas (funciones predefinidas), tal que las salidas de cada unidad se utilicen como entradas de otras unidades, de tal forma que al final se obtenga la relación entrada-salida global deseada. En resumen, el proceso de programación bajo el paradigma funcional consiste en construir funciones como conjuntos anidados de otras funciones más simples.

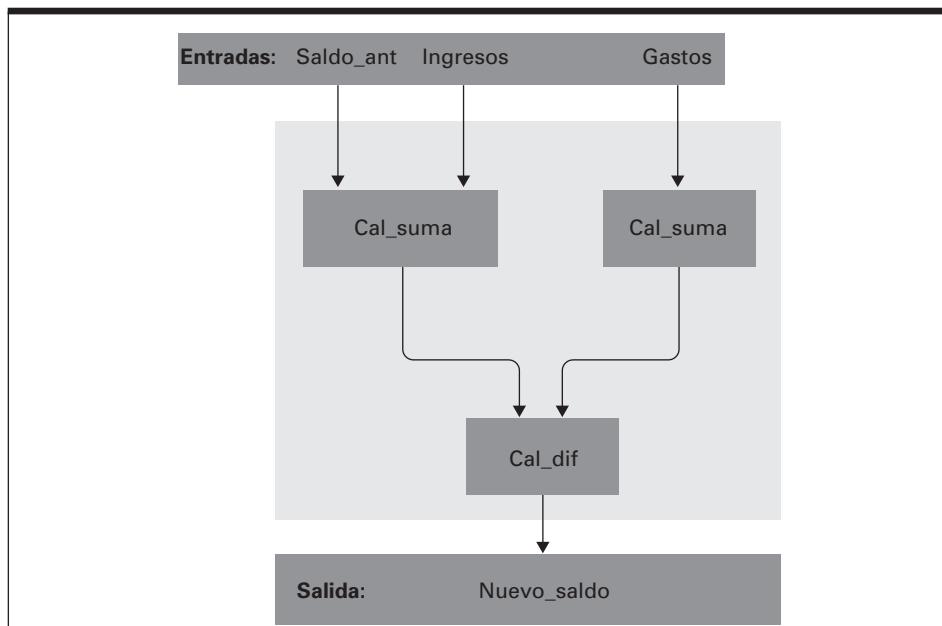
Veamos un ejemplo. La Figura 6.3 muestra cómo se puede construir una función que calcule el saldo de nuestra cuenta corriente a partir de dos funciones más simples. Una de ellas, de nombre `Cal_suma`, acepta valores como entrada y genera la suma de esos valores como salida. La otra, denominada `Cal_dif`, acepta dos valores de entrada y calcula su diferencia. La estructura mostrada en la Figura 6.3 se puede representar en el lenguaje de programación LISP (uno de los principales lenguajes de programación funcionales) mediante la expresión

```
(Cal_dif (Cal_suma Saldo_ant Ingresos) (Cal_suma Gastos))
```

La estructura anidada de esta expresión (como indican los paréntesis) refleja el hecho de que las entradas a la función `Cal_dif` son generadas aplicando `Cal_suma` dos veces. La primera ejecución de `Cal_suma` genera el resultado de sumar todos los `Ingresos` al `Saldo_ant`. La segunda ejecución de `Cal_suma` calcula el total de todas los `Gastos`. Entonces, la función `Cal_dif` utiliza estos resultados para obtener el nuevo saldo de nuestra cuenta bancaria.

Para entender mejor la distinción entre los paradigmas funcional e imperativo, vamos a comparar el programa funcional para calcular el saldo de una cuenta corriente con el siguiente programa de pseudocódigo obtenido ajustándose al paradigma imperativo:

Figura 6.3 Una función para calcular el saldo de una cuenta bancaria construida a partir de funciones más simples.



```

Total_ingresos ← suma de todos los Ingresos
Saldo_temp ← Saldo_ant + Total_ingresos
Total_gastos ← suma de todos los Gastos
Saldo ← Saldo_temp - Total_gastos

```

Observe que este programa imperativo consta de varias sentencias, cada una de las cuales solicita que se realice un cálculo y que el resultado se almacene para un uso posterior. A diferencia de esto, el programa funcional consta de una única sentencia, en la que el resultado de cada cálculo se canaliza de forma inmediata al siguiente. En cierto sentido, el programa imperativo es análogo a un conjunto de fábricas, donde cada fábrica convierte sus materias primas en productos que se guardan en almacenes. Posteriormente, los productos de estos almacenes son suministrados a otras fábricas a medida que los van necesitando. Pero el programa funcional es análogo a un conjunto de fábricas que están coordinadas, de manera que cada una solo fabrica aquellos productos que han sido solicitados por otras fábricas y envía estos productos a sus destinos de forma inmediata sin pasar por un almacenamiento intermedio. Esta eficiencia es una de las ventajas proclamadas por los defensores del paradigma funcional.

Otro paradigma de programación más (y el más importante actualmente en el desarrollo de software) es el **paradigma orientado a objetos**, que está asociado con el proceso de programación denominado **programación orientada a objetos** (*Object-Oriented Programming*). De acuerdo con este paradigma, un sistema software se ve conceptualmente como un conjunto de unidades, denominadas **objetos**, cada uno de las cuales es capaz de llevar a cabo las acciones que le afectan directamente, así como de solicitar acciones a otros objetos. De forma conjunta, estos objetos interactúan para resolver el problema que tengamos entre manos.

Como ejemplo práctico de la técnica de orientación a objetos, considere la tarea de desarrollar una interfaz gráfica de usuario. En un entorno orientado a objetos, los iconos que aparecen en la pantalla se implementarían como objetos. Cada uno de estos objetos incluiría un conjunto de procedimientos (denominados **métodos** en la jerga de la orientación a objetos) que describirían cómo debe responder el objeto a diversos posibles eventos, como por ejemplo al evento de ser seleccionado por un clic del botón del ratón o al evento de ser arrastrado con el ratón a través de la pantalla. De este modo, el sistema completo se construiría como un conjunto de objetos, cada uno de los cuales sabe cómo responder a los eventos relacionados con él.

Para comparar el paradigma orientado a objetos con el paradigma imperativo, considere un programa en el que se emplee una lista de nombres. En el paradigma imperativo tradicional, esta lista sería simplemente un conjunto de datos. Cualquier unidad de programa que acceda a la lista deberá contener los algoritmos para realizar las manipulaciones requeridas. Sin embargo, en la técnica de orientación a objetos la lista se construiría como un objeto que estaría compuesto por la propia lista y un conjunto de métodos para manipular la lista (esto podría incluir procedimientos para insertar una entrada en la lista, borrar una entrada de la lista, detectar si la lista está vacía y ordenar la lista). Esto quiere decir que cualquier otra unidad de programa que necesitara manipular la lista no contendría ningún algoritmo para realizar las tareas pertinentes. En lugar de ello, haría uso de los procedimientos proporcionados por el objeto. En cierto sentido, en lugar de ordenar la lista, como haríamos en el paradigma

imperativo, lo que una unidad de programa haría en el paradigma de la orientación a objetos sería pedir a la lista que se ordene a sí misma.

Aunque hablaremos con más detalle del paradigma orientado a objetos en la Sección 6.5, su importancia en el panorama actual del desarrollo software nos obliga a incluir el concepto de clase en esta presentación. Con este objetivo, recuerde que un objeto puede estar formado por datos (tal como una lista de nombres) y por un conjunto de métodos para la realización de actividades (como por ejemplo insertar nuevos nombres en la lista). Estas características deben ser descritas mediante sentencias incluidas en el programa que escribamos. Esta descripción de las propiedades de un objeto se denomina **clase**. Una vez construida una clase, puede aplicarse cada vez que haga falta un objeto con dichas características. De este modo, puede haber múltiples objetos basados en la misma clase, es decir, construidos a partir de esa clase. Al igual que sucede con dos gemelos idénticos, estos objetos serían entidades distintas, pero tendrían las mismas características porque han sido construidos a partir de la misma plantilla (la misma clase). (Un objeto que está basado en una clase concreta se dice que es una **instancia** de esa clase.)

La razón de que el paradigma de orientación a objetos haya ganado popularidad es, precisamente, porque los objetos son unidades bien definidas, cuyas descripciones están aisladas en clases reutilizables. De hecho, los defensores de la programación orientada a objetos argumentan que el paradigma de la orientación a objetos proporciona un entorno natural para la técnica de desarrollo software basada en la utilización de "bloques de código reutilizables". Esos expertos tienen la visión de que deberíamos poder disponer de bibliotecas software de clases predefinidas, a partir de las cuales pudieran construirse nuevos sistemas software de la misma forma en que muchos productos tradicionales se construyen a partir de componentes disponibles en el mercado. La construcción y expansión de dicho tipo de bibliotecas es un proceso que está actualmente en marcha, como veremos en el Capítulo 7.

Para terminar, hay que observar que los métodos dentro de un objeto son, en esencia, pequeñas unidades de programa de tipo imperativo. Es decir que la mayoría de los lenguajes de programación basados en el paradigma orientado a objetos contienen muchas de las características que podemos encontrar en los lenguajes imperativos. Por ejemplo, el popular lenguaje orientado a objetos C++ fue desarrollado añadiendo características de orientación a objetos al lenguaje imperativo conocido como C. Además, puesto que Java y C# son derivados de C++, también ellos han heredado ese núcleo de tipo imperativo. En las Secciones 6.2 y 6.3 exploraremos muchas de estas características imperativas y, al hacerlo, hablaremos de una serie de conceptos que son fundamentales para gran parte del software orientado a objetos utilizado hoy día. Después, en la Sección 6.5, consideraremos aquellas características que son exclusivas del paradigma orientado a objetos.

Cuestiones y ejercicios

1. ¿En qué sentido es independiente de la máquina un programa escrito en un lenguaje de tercera generación? ¿En qué sentido continúa siendo dependiente de la máquina?



apéndice

D

Lenguajes de programación de alto nivel

Este apéndice proporciona un breve resumen sobre cada uno de los lenguajes utilizados como ejemplos en el Capítulo 6.

Ada

El lenguaje Ada, denominado así en honor de Augusta Ada Byron (1815–1851), que fue una defensora de Charles Babbage y la hija del poeta Lord Byron, fue desarrollado por iniciativa del Departamento de Defensa de Estados Unidos en un intento de obtener un único lenguaje de propósito general que satisfaciera todas sus necesidades de desarrollo de software. Uno de los principales objetivos durante el diseño de Ada fue incorporar características para programar sistemas de computadora en tiempo real que pudieran utilizarse como parte de máquinas de mayor tamaño, como por ejemplo sistemas de guiado de misiles, sistemas de control medioambiental dentro de edificios y sistemas de control para automóviles y pequeños electrodomésticos. Ada contiene por tanto características para expresar actividades en entornos de procesamiento paralelo, además de una serie de técnicas de utilidad para la gestión de casos especiales (denominados excepciones) que pueden surgir dentro del entorno de aplicación. Aunque originalmente se desarrolló como un lenguaje imperativo, las versiones más recientes de Ada han adoptado el paradigma de la orientación a objetos.

El diseño del lenguaje Ada ha puesto siempre el énfasis en aquellas características que permiten un desarrollo eficiente de software fiable, una característica que queda ilustrada por el hecho de que todo el software de control interno del avión Boeing 777 fue escrito en Ada. Esta es también una de las principales razones por las que se utilizó Ada como punto de partida a la hora de desarrollar el lenguaje SPARK, como se ha indicado en el Capítulo 5.

C

El lenguaje C fue desarrollado por Dennis Ritchie en los Bell Laboratories a principios de la década de 1970. Aunque diseñado originalmente como un lenguaje para el desarrollo de software de sistemas, C ha conseguido una gran

popularidad en toda la comunidad de programadores y ha sido estandarizado por el instituto ANSI.

C se concibió originalmente como un lenguaje situado un escalón por encima del lenguaje máquina. En consecuencia, su sintaxis es muy seca, si la comparamos con otros lenguajes de alto nivel que emplean palabras en inglés completas para expresar algunas de las primitivas que en C se representan mediante símbolos especiales. Esta sequedad permite una representación eficiente de algoritmos complejos, lo cual es una de las principales razones de la popularidad del lenguaje C. (A menudo, una representación concisa es más legible que otra muy larga.)

C++

El lenguaje C++ fue desarrollado por Bjarne Stroustrup en los Bell Laboratories como una versión ampliada del lenguaje C. El objetivo era obtener un lenguaje compatible con el paradigma de la orientación a objetos. Actualmente, C++ no es solo uno de los principales lenguajes orientados a objetos, sino que se ha utilizado como punto de partida para el desarrollo de otros dos de los principales lenguajes orientados a objetos: Java y C#.

C#

El lenguaje C# fue desarrollado por Microsoft como una herramienta para el entorno .NET, que es un sistema muy completo para el desarrollo de software de aplicación para máquinas que ejecuten el software de sistemas de Microsoft. El lenguaje C# es muy similar a C++ y Java. De hecho, la razón de que Microsoft introdujera C# como un lenguaje diferente, no es porque se trate de un lenguaje verdaderamente nuevo, sino que como lenguaje diferente que es, Microsoft podía personalizar características específicas del lenguaje sin preocuparse de cumplir con los estándares que afectaban a otros lenguajes y sin preocuparse tampoco por los derechos de propiedad intelectual de otras empresas. Por tanto, la novedad de C# radica en su papel como lenguaje principal para el desarrollo de software que utilice el entorno .NET. Con el respaldo de Microsoft, C# y el entorno .NET prometen ser actores relevantes en el campo del desarrollo software durante los años venideros.

Fortran

FORTRAN es el acrónimo de FORmula TRANslator (traductor de fórmulas). Este lenguaje fue uno de los primeros lenguajes de alto nivel que se desarrollaron (fue presentado en 1957) y fue también uno de los primeros lenguajes en obtener una amplia aceptación dentro de la comunidad de los profesionales de la computación. A lo largo de los años, su descripción oficial ha sufrido numerosas ampliaciones, lo que significa que el lenguaje FORTRAN actual es muy diferente del original. De hecho, estudiando la evolución de FORTRAN podemos ver los efectos que las sucesivas investigaciones han ido teniendo en el diseño de los lenguajes de programación. Aunque originalmente se diseñó como lenguaje imperativo, las versiones más recientes de FORTRAN incluyen ahora muchas características de orientación a objetos. FORTRAN continúa siendo un lenguaje muy popular dentro de la comunidad científica. En particu-

lar, muchos paquetes estadísticos y de análisis numérico están escritos en FORTRAN y muy probablemente sigan estandarizándolo en el futuro.

Java

Java es un lenguaje orientado a objetos desarrollado por Sun Microsystems a principios de la década de 1990. Sus diseñadores tomaron prestadas numerosas ideas de C y C++. El entusiasmo por Java se debe no al propio lenguaje, sino a su implementación universal y al impresionante número de plantillas prediseñadas que hay disponibles en el entorno de programación Java. Lo de implementación universal significa que un programa escrito en Java puede ejecutarse de forma eficiente en un amplio rango de máquinas, mientras que la disponibilidad de plantillas significa que puede desarrollarse software complejo con relativa facilidad. Por ejemplo, las plantillas tipo applet y servlet permiten simplificar el desarrollo de software para la World Wide Web.

Para convertir un programa en lenguaje máquina, necesitamos un software de traducción. Dicho programa puede ser un **intérprete** (un programa que traduce y transmite cada sentencia de forma individual, del mismo modo que en las Naciones Unidas se traduce un discurso del ruso al español), o un **compilador** (un programa que traduce el programa completo antes de pasarlo a la computadora, del mismo modo que un estudiante puede traducir la novela **Guerra y Paz** del ruso al español). La mayoría de los traductores de C++ son compiladores, ya que los programas compilados tienden a ejecutarse más deprisa que los interpretados.

Cualquier software de compilación actual es mucho más que un simple compilador. Es un **entorno de programación integrado** que incluye un editor de textos, un compilador, un **depurador (debugger)** para simplificar el proceso de localización y corrección de errores, y otras utilidades de programación. Los **errores sintácticos** (violaciones de las reglas gramaticales del lenguaje de programación) suelen mostrarse automáticamente tan pronto como se teclean en el editor. Los **errores lógicos** (problemas con la estructura lógica que provocan diferencias entre lo que se supone que el programa debe hacer y lo que realmente hace) no siempre son sencillos de detectar. Ésta es la razón por la que la depuración y la verificación son fases que se pueden llevar una gran parte del tiempo de desarrollo de un programa.

Si un carácter, una pausa, del conjuro no está en la forma adecuada, la magia no funciona.

—Frederick Brooks,
en *The Mythical Man-Month*

Lenguajes de programación y metodologías

C++ es uno de los cientos de lenguajes de programación utilizados hoy en día. Algunos son herramientas para los programadores profesionales que se encargan de desarrollar el software que el resto de mortales utilizamos. Otros están orientados a ayudar a los estudiantes a aprender los fundamentos de la programación. Un tercer grupo permite que los usuarios de las computadoras automatizan tareas repetitivas y optimicen aplicaciones software. Desde los primeros días de la informática, los lenguajes de programación han evolucionado hacia una forma más simple de comunicación entre las personas y las máquinas.

Lenguaje máquina y lenguaje ensamblador

Cada computadora tiene un lenguaje nativo, un **lenguaje máquina**. Existen similitudes entre las distintas clases de lenguajes máquina: todos ellos tienen instrucciones para efectuar las cuatro operaciones aritméticas básicas, para comparar pares de números, instrucciones para formar bucles, etc. Pero, al igual que el español y el francés, cada uno de estos lenguajes máquina son lenguajes diferentes, y las máquinas basadas en uno de ellos no pueden entender los programas escritos en otro.

Desde el punto de vista de las máquinas, el lenguaje máquina es binario. Las instrucciones, las localizaciones de memoria, los números y los caracteres están representados por cadenas de ceros y unos. Como los números binarios son complicados de leer, los programas en lenguaje máquina suelen mostrarse convertidos a decimal (base 10), **hexadecimal** (base 16) o cualquier otro sistema de numeración. Aun así, estos programas siempre han sido difíciles de escribir, leer y depurar.

Guía visual

Programación en C++

```

Microsoft Developer Studio : Game : Game.cpp
File Edit View Insert Build Tools Window Help
File View Insert Build Tools Window Help
Game - Win32 Release
Game.cpp Game.h Game.h Game.cpp
Turn();
EndGame();

return EXIT_SUCCESS;
}

void
StartGame()
{
    cout << "Welcome to the guessing game. I'll pick a number
    << "between 1 and 100 and you try to guess what it is.
    << You get 7 tries." << endl << endl;

    // calculate a random number between 1 and 100
}

```

Figura 14.1a Ha decidido convertir el algoritmo anterior en un programa útil. Una vez escrito, puede escribirlo en la ventana del editor. Éste sangra automáticamente las sentencias a medida que las escribe, de modo que resulte fácil ver la estructura lógica del programa. El editor también marca los errores sintácticos.

Figura 14.1b Se ejecuta el programa para verificar su lógica. Cuando se comprueba con varios intentos incorrectos, se ve que no se detiene tras introducir los siete intentos. Y cuando se introduce el valor adecuado, también falla porque no termina y vuelve a pedir otro valor.

```

app(36) : error C2065: 'you' : undeclared identifier
MS-DOS Prompt, F:\Windows\Temp\file1 /P
Ready
Linking...
Game.exe = 0 error(s), 0 warning(s)

```

```

if (Guess == N)
{
    cout << "1"
    Counter =
}
else if (Guess
    cout << "I"
else cout << "A"

Counter++
} while (Counter <
}

Linking...
Game.exe = 0 error(s), 0 warning(s)

```

Figura 14.1c Se detecta el error lógico en la sentencia Counter—, que decrementa el contador en 1. Debería decir Counter++, para incrementar la variable en una unidad. Se corrige el error y se vuelve a ejecutar el programa para comprobar de nuevo su funcionamiento. Como cualquier otro, este programa puede pasar por varias fases de comprobación, depuración y refinado antes de que funcione de forma satisfactoria.

Con la invención del **ensamblador** (un lenguaje funcionalmente similar al lenguaje máquina pero más sencillo de escribir, leer y comprender por las personas) el proceso de programación se hizo más sencillo. En este lenguaje, los programadores utilizan códigos alfabéticos que se corresponden con las instrucciones numéricas de la máquina. Por ejemplo, la sentencia ensamblador para restar podría ser SUB. Desde luego, SUB no significa nada para la computadora, la cual sólo responde ante comandos del tipo 10110111. Para establecer un lazo de unión entre el programador y la computadora, un programa llamado **ensamblador** traduce cada instrucción de este lenguaje en la sentencia máquina correspondiente. Sin conocer nada mejor, la computadora actúa como su propio traductor.

Debido a las claras ventajas del ensamblador, ya son muy pocos los programadores que utilizan el lenguaje máquina. Pero la codificación en ensamblador aún se considera programación a bajo nivel, es decir, obliga al programador a pensar al nivel de la máquina y a incluir una enorme cantidad de detalles en cada programa. Los lenguajes ensamblador y máquina son **lenguajes de bajo nivel**, los cuales conllevan procesos repetitivos, tediosos y muy propensos a los errores. Para complicar aún más las cosas, cualquier programa escrito en uno de estos lenguajes deben ser reescritos por completo antes de poder utilizarlos en una computadora con un lenguaje máquina diferente. Muchos programadores siguen usando el ensamblador para escribir partes de videojuegos y otras aplicaciones en las que la velocidad y la comunicación con el hardware es un factor crítico. Pero la mayoría de los programadores de hoy en día piensan y escriben en un nivel superior.

La programación de una computadora es una forma de arte, como la **poesía** o la **música**.

—Donald E. Knuth, autor de *The Art of Computer Programming*

Lenguajes de alto nivel

Los **lenguajes de alto nivel**, que están a medio camino entre el lenguaje natural de los humanos y los lenguajes máquina, fueron desarrollados a principio de la década de los 50 para simplificar y perfilar el proceso de programación. Lenguajes como FORTRAN y COBOL permiten que los científicos, ingenieros y gente de negocio escriban programas usando una terminología y notación familiar en lugar de las enigmáticas instrucciones máquina. En la actualidad, los programadores pueden escoger entre cientos de lenguajes de alto nivel.

Los intérpretes y los compiladores traducen los programas de alto nivel en lenguaje máquina. Una vez interpretada o compilada, una sentencia de uno de estos lenguajes se transforma en varias instrucciones máquina. Un lenguaje de alto nivel oculta al programador la mayoría de los detalles oscuros de las operaciones máquina. Como resultado de ello, resulta más sencillo centrarse en la lógica básica del programa, es decir, en la idea principal.

Además de ser más sencillos de escribir y depurar, los programas de alto nivel tienen la ventaja de poder transportarse de una máquina a otra. Un código escrito en C estándar puede ser compilado y ejecutado en cualquier computadora que disponga de este compilador. El mismo concepto se aplica a los programas escritos en Java, Basic, FORTRAN o COBOL.

Transportar un programa de una máquina a otra no siempre es tan sencillo. La mayoría de los programas de alto nivel deben ser reescritos para que se ajusten a los diferentes dispositivos hardware, compiladores, sistemas operativos e interfaces de usuario. Por ejemplo, cuando se pasa de un programa de su versión Windows a Macintosh, es preciso reescribir alrededor del 20 por ciento del código, o viceversa. Aun así, los programas de alto nivel son más portables que los escritos en ensamblador o lenguajes máquina.

De los cientos de lenguajes de alto nivel que se han desarrollado, algunos se han convertido en muy populares debido a su amplio uso:

- **FORTRAN** (*Formula Translation*), el primer lenguaje de alto nivel comercial, fue diseñado por IBM en la década de los 50 para resolver problemas científicos y de ingeniería. En la actualidad, muchos científicos utilizan las versiones más modernas de este lenguaje.
- **COBOL** (*Common Business Oriented Language*) fue desarrollado en 1960 cuando el gobierno norteamericano solicitó un nuevo lenguaje orientado a los negocios y a los problemas derivados del procesamiento de datos. Los programadores en COBOL aún trabajan en muchas empresas de procesamiento de datos de todo el mundo.
- **LISP** (*List Processing*) fue desarrollado en el MIT a finales de los 50 para procesar datos no numéricos como caracteres, palabras y otros símbolos. LISP es ampliamente utilizado en la investigación de inteligencia artificial, en parte porque resulta más sencillo escribir programas en este lenguaje que en otros.
- **Basic** (*Beginner's All-purpose Symbolic Instruction Code*; también puede verse como BASIC) fue desarrollado a mediados de los 60 como una alternativa al FORTRAN interactiva y fácil de aprender para programadores principiantes. Antes del Basic, cualquier estudiante tenía que enviar el programa, esperar durante varias horas el resultado de la compilación y repetir el proceso cada vez que se detectaba un error. Como el Basic era interpretado línea a línea en lugar de compilarse, podía ofrecer un resultado instantáneo a medida que los estudiantes introducían los comandos en sus terminales. Cuando aparecieron los PC, Basic disfrutó de una popularidad sin precedentes entre los estudiantes, fanáticos y programadores. Con el tiempo, Basic se ha convertido en una potente y moderna herramienta de programación para programadores noveles y profesionales. True Basic es una versión moderna desarrollada por los padres del lenguaje original. La versión de Basic para Windows más popular hoy en día (y, de hecho, el lenguaje de programación más popular jamás creado) es el **Visual Basic** de Microsoft. REALBasic es una versión para Macintosh muy parecida a Visual Basic.
- **Pascal** (llamado así en honor al matemático, inventor, filósofo y místico francés del siglo XVII) fue desarrollado a comienzos de los 70 como una alternativa al BASIC para los estudiantes que estaban aprendiendo a programar. Este lenguaje fue diseñado para fomentar la programación estructurada, una técnica descrita en la siguiente sección. Pascal ya casi no es utilizado por la comunidad de programadores profesionales.
- **C** fue inventado en los laboratorios Bell a comienzos de los 70 como una herramienta para programar sistemas operativos como UNIX. C es un lenguaje complejo bastante difícil de aprender, aunque su potencia, flexibilidad y eficiencia hacen de él, en cualquiera de sus variantes, el lenguaje preferido de los profesionales que programan computadoras personales.
- **C++** es el lenguaje empleado en nuestro ejemplo de guía visual. C++ es una variante de C que se beneficia de una metodología de programación moderna llamada programación orientada a objetos, la cual se describe con detalle en este mismo capítulo.
- **C#** es un popular lenguaje exclusivo de Windows muy similar a C++.
- **Java** es un lenguaje de programación moderno desarrollado por Sun Microsystems, muy parecido a C++ aunque más sencillo y fácil de aprender. Java se dis-

tingue por la generación de *applets* para la Web que funcionan en múltiples plataformas.

- **J++** es un lenguaje de programación como Java propiedad de Microsoft y adaptado para la plataforma Windows.
- **ActiveX** es un lenguaje de Microsoft diseñado específicamente para la creación de componentes web similares a los *applets* Java.
- **Python** es un lenguaje similar a Java muy popular entre los programadores Linux.
- **Ada** (llamado así en honor de Ada King, la primera programadora conocida cuya vida se comentó brevemente en el Capítulo 1) es un lenguaje basado en Pascal. Fue desarrollado a comienzos de la década de los 70 por el Departamento de Defensa de los Estados Unidos. Ada nunca superó los muros de las instalaciones militares.
- **PROLOG** (*Programming Logic*) es un popular lenguaje para la programación de inteligencia artificial. Como su nombre sugiere, PROLOG está diseñado para trabajar con relaciones lógicas entre hechos.
- **LOGO** es un dialecto de LISP especialmente diseñado para niños.

Programación estructurada

Un lenguaje de programación puede ser una potente herramienta en manos de un programador adiestrado. Pero las herramientas por sí solas no son garantía de calidad; los mejores programadores disponen de técnicas específicas para obtener el máximo rendimiento de sus programas. En la corta historia de la programación de computadoras, los informáticos han desarrollado nuevas metodologías que han conseguido hacer más productivos a los programadores y más fiables a los programas.

Por ejemplo, los informáticos de finales de los 60 reconocían que la mayoría de los programas FORTRAN y BASIC estaban repletos de sentencias GoTo, las cuales transfieren el control a otras partes del código. La estructura lógica de un programa con sentencias GoTo puede llegar a parecerse a una intrincada red de araña. Cuanto mayor es ese programa, mayor es el laberinto lógico y la posibilidad de cometer errores. Cada rama de un programa representa un cabo suelto que el programador debe pasar por alto.

En un intento por superar estos problemas, los informáticos desarrollaron la **programación estructurada**, una técnica para hacer más productivo y sencillo el proceso de programación. Un programa estructurado no depende de sentencias GoTo para controlar su flujo de ejecución. En lugar de ello, está construido a base de pequeños programas, llamados **módulos** o **subprogramas**, los cuales a su vez también están construidos a partir de otros módulos más pequeños. El programador combina estos subprogramas mediante las tres estructuras de control básicas: secuencia, repetición y selección. Un programa está bien estructurado si cumple las siguientes reglas:

- Esta construido sobre la base de módulos lógicamente coherentes.
- Los módulos están ordenados jerárquicamente.
- Es sencillo y fácil de leer.

Pascal y Ada fueron diseñados para fomentar la programación estructurada y disuadir del «código spaghetti». El éxito de estos lenguajes indujo a los informáticos a desarrollar versiones de Basic y FORTRAN que siguieran los patrones de la programación estructurada.