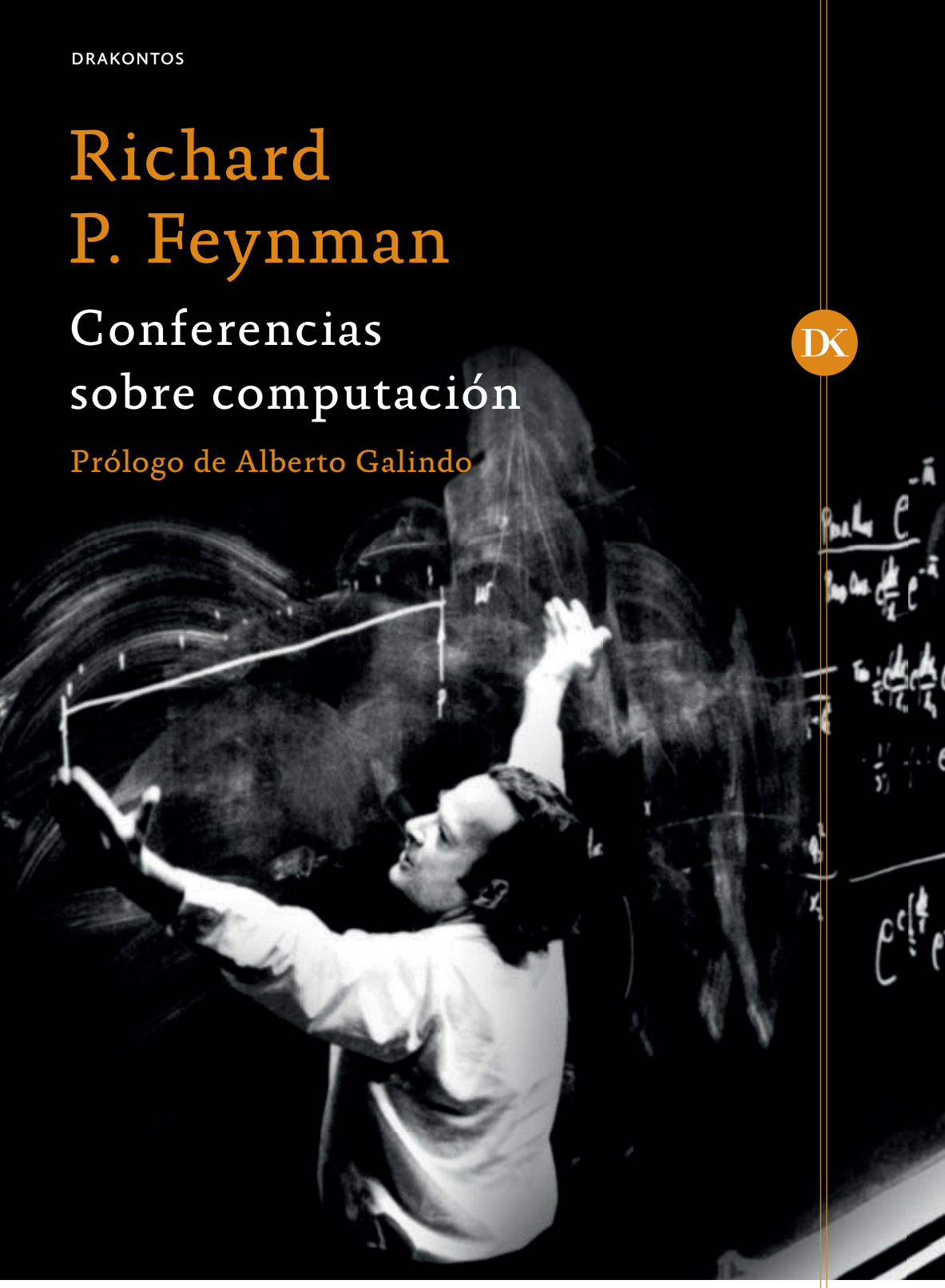
DRAKONTOS

Richard P. Feynman

Conferencias sobre computación

Prólogo de Alberto Galindo



CRÍTICA

Conferencias sobre computación

Richard P. Feynman

Prólogo de Alberto Galindo

Edición de Anthony J. G. Hey
y Robin W. Allen

CRÍTICA
BARCELONA

Primera edición: octubre de 2003
Primera edición en esta nueva presentación: abril de 2018

Conferencias sobre computación
Richard P. Feynman

Título original: *Feynman Lectures on Computation*

Av. Diagonal, 662-664, 08034 Barcelona (España)
Crítica es un sello editorial de Editorial Planeta, S. A.

editorial@ed-critica.es
www.ed-critica.es

ISBN: 978-84-17067-92-2
Depósito legal: B. 6013 - 2018
2018. Impreso y encuadernado en España

El papel utilizado para la impresión de este libro es 100% libre de cloro
y está calificado como papel ecológico.

1

Introducción a los computadores

Los computadores pueden hacer muchas cosas. Pueden sumar millones de números en un abrir y cerrar de ojos. Pueden burlarse de los grandes maestros del ajedrez. Pueden dirigir las armas hacia su objetivo. Pueden reservarnos un asiento en un avión entre una monja que toque la guitarra y un catedrático de física que no fume. Algunos pueden incluso tocar los bombos. ¡Bastante variado! Por lo tanto, si vamos a hablar de computadores mejor decidimos ya cuáles son los que vamos a tratar y cómo lo vamos a hacer.

De hecho, no vamos a dedicar mucho tiempo a ninguna máquina en particular. La razón es que una vez que se ha llegado a entender los entresijos de los computadores encontramos que, como las personas, todos tienden a ser más o menos iguales. Pueden diferir en sus funcionalidades y en la naturaleza de sus entradas y salidas —uno puede producir música y otro hacer un dibujo, mientras que uno se puede iniciar desde un teclado, otro puede serlo por el par de fuerzas en las ruedas de un automóvil—, pero en el fondo todos son muy parecidos. Nos ocuparemos, por lo tanto, solamente de sus entrañas, sin suponer nada de la estructura de su entrada-salida (I/O) o de cómo la información llega y sale de la máquina. De lo único que nos preocuparemos es de que, cualquiera que sea el medio por el que llega la entrada y cualquiera que sea el resultado, tanto la entrada como el resultado estén en forma digital. Por digital entiendo números binarios: unos y ceros.

¿Cómo es el interior de un computador? Está construido de un conjunto de elementos básicos simples. Estos elementos no son nada especiales, pueden ser válvulas de control o cuentas en el alambre de un ábaco, y hay muchas posibilidades para elegir el conjunto de elementos básicos. Lo único que importa es que se puedan utilizar para construir

todo lo que deseamos. ¿Cómo están organizados? De nuevo hay muchas posibilidades para elegir; la estructura relevante probablemente se determina por consideraciones tales como la velocidad, la disipación energética, la estética y lo que se tenga disponible. Visto de esta forma, la variedad de computadores es un poco como la variedad de casas: una mansión en Beverly Hills puede parecer completamente diferente de un garaje en Yonkers, pero ambos están construidos de lo mismo, ladrillos, cemento, madera y sudor, sólo que la mansión tiene más de todo y ordenado de manera diferente, de acuerdo con las necesidades de los propietarios. Pero en el fondo son muy parecidos.

Seamos un poco abstractos por un momento y preguntémonos: *¿cómo* se conectan *qué* conjunto de elementos para que hagan la *mayor parte* de las cosas? Ésta es una pregunta profunda cuya respuesta es, de nuevo, que hasta cierto punto no importa. Una vez que se tiene un computador que puede hacer unas cuantas cosas —estrictamente hablando, uno que tiene un «conjunto suficiente» de procedimientos básicos— ese computador puede hacer básicamente lo mismo que cualquier otro. Esto es vagamente la base del gran principio de «universalidad». «¡Ah! —gritas—. ¡Mi calculadora de bolsillo no puede simular las manchas rojas de Júpiter como un conjunto de supercomputadores Cray!» Pues sí, sí puede: habría que modificar algunos circuitos, tendríamos que añadirle memoria y sería terriblemente lento, pero si se espera lo suficiente podría reproducir cualquier cosa que los Crays puedan hacer. En general, supongamos que tenemos dos computadores, A y B, y que sabemos todo acerca de A, la forma en que trabaja, sus «reglas de transición entre estados» y cosas así. Supongamos que el computador B es capaz simplemente de *describir* el estado de A. Entonces podemos utilizar B para simular el proceso de A describiendo sus transiciones sucesivas; en otras palabras, B imitará a A. Podrá tardar una eternidad si B es muy elemental y A muy sofisticado, pero al final podrá hacer todo lo que haga A. Más adelante en el curso demostraremos esta afirmación diseñando tal computador B, conocido como la máquina de Turing.

Veamos la universalidad de otra forma. El lenguaje nos proporciona una útil fuente de analogías. Dejadme que os pregunte esto: ¿cuál es el *mejor* lenguaje para describir algo? Digamos, un vehículo de gasolina de cuatro ruedas. Desde luego, la mayoría de las lenguas, al menos en Occidente, tienen una palabra sencilla para ello: nosotros decimos «automóvil», en inglés se dice «car», en francés «voiture», y así sucesivamente. Sin embargo, habrá algunos idiomas en los que no se haya creado una palabra para el «automóvil», y los que hablen en ese idioma tendrán que re-

currir a una descripción seguramente larga y complicada utilizando elementos lingüísticos básicos. Ninguna de estas dos descripciones es intrínsecamente «mejor» que la otra, porque ambas cumplen su cometido y sólo se distinguen por su eficiencia. No tenemos por qué introducir la democracia al nivel de las palabras; podemos bajar a ras de los alfabetos. ¿Cuál es, por ejemplo, el mejor alfabeto para la lengua inglesa? Es decir, ¿por qué tenemos que quedarnos con las 26 letras usuales? Todo lo que se puede hacer con éstas, se puede hacer sólo con tres símbolos —punto, raya y espacio del código Morse; o dos— un cifrador baconiano que representa desde la A hasta la Z con números binarios de cinco dígitos. Vemos, pues, que podemos elegir nuestro conjunto básico de elementos con mucha libertad y nuestra elección realmente afecta a la eficiencia de nuestro lenguaje y, por lo tanto, al tamaño de nuestros libros: no existe el «mejor» lenguaje o alfabeto —cada uno es lógicamente universal y cada uno puede imitar a los demás—. Volviendo a la computación, la universalidad establece, de hecho, que el conjunto de tareas complicadas que se pueden realizar utilizando un conjunto «suficiente» de procedimientos básicos es independiente de la estructura específica y detallada del conjunto básico.

Para que los computadores actuales realicen una tarea compleja necesitamos una descripción precisa y completa de cómo realizar esa tarea en términos de una secuencia de procedimientos básicos sencillos —el software— y necesitamos una máquina para que realice esos procedimientos en un orden específico —esto es, el «hardware»—. Estas instrucciones tienen que ser exactas y sin ambigüedad. Las personas no tienen que decirse unas a otras *exactamente* lo que quieren expresar; no es necesario porque el contexto, el lenguaje corporal, la familiaridad con el interlocutor y demás recursos nos permiten «llenar los huecos» y resolver cualquier ambigüedad en lo que queremos decir. Los computadores, sin embargo, aún no son capaces de «seguir» lo que se está diciendo de la forma que lo hace una persona. A éstos hay que decirles escrupulosamente todos los detalles de lo que tienen que hacer. Quizá algún día las máquinas sean capaces de entender descripciones aproximadas de tareas, pero mientras tanto tenemos que ser muy quisquillosos acerca de cómo le decimos a los computadores que hagan las cosas.

Veamos cómo se pueden construir instrucciones complejas a partir de un conjunto de elementos rudimentarios. Obviamente, si el conjunto de instrucciones, llamémosle *B,* es muy sencillo un proceso complejo necesitará una gran cantidad de descripción y los «programas» resultantes serán largos y complicados. Supongamos, por ejemplo, que queremos que

nuestro computador realice cálculos numéricos, pero resulta que el conjunto *B* no incluye la multiplicación como una operación independiente. Si le decimos a nuestra máquina que multiplique 3 por 35 nos dirá «¿qué?». Pero supongamos que *B* sí tiene la suma; si piensas un poco podrás ver que puedes conseguir que multiplique simplemente sumando muchas veces —en este caso suma 35 tres veces—. Sin embargo, los programas escritos con *B* serán más claros si aumentamos el conjunto con la instrucción independiente «multiplicar», *definida* por un grupo de instrucciones básicas de *B* que realiza la multiplicación. Entonces, cuando deseemos multiplicar dos números diremos «computador, 3 por 35» y ahora reconocerá la palabra «por», que es simplemente realizar varias sumas y las llevará a cabo. La máquina descompone estas situaciones complejas en sus componentes básicos, salvándonos de quedar atascados continuamente en conceptos de nivel más bajo. Los procedimientos complejos se construyen así nivel a nivel. Un proceso muy parecido tiene lugar en la vida ordinaria cuando uno expresa con una sola palabra un conjunto de ideas y las relaciones entre ellas. Para referirme a esas ideas y a sus interconexiones utilizamos simplemente una palabra y así evitamos tener que volver a reconstruir todo a partir de los conceptos del nivel más bajo. Los computadores son objetos tan complicados que generalmente son necesarias este tipo de ideas simplificadoras, y un buen diseño es esencial si uno no quiere quedarse completamente perdido en los detalles.

Empezaremos construyendo un conjunto de procedimientos elementales y examinaremos cómo realizar operaciones tales como sumar dos números o transferir dos números de un lugar de la memoria a otro. Luego subiremos de nivel, al siguiente orden de complejidad, y utilizaremos estas instrucciones para hacer operaciones tales como multiplicar, y así sucesivamente. No iremos muy lejos en esta jerarquía, pero si deseamos ver hasta dónde se puede llegar, el artículo sobre sistemas operativos de P. J. Denning y R. L. Brown (*Scientific American*, septiembre de 1984, pp. 96-104) establece ¡trece niveles! Va desde el nivel 1, circuitos electrónicos —registros, puertas, buses— hasta el número 13, el shell del sistema operativo, que interpreta la programación del usuario. Componiendo instrucciones de una forma jerárquica se pasa de las transferencias básicas de unos y ceros del nivel 1 hasta el nivel 13, en el cual hay órdenes para hacer aterrizar un avión en un simulador o para comprobar si un número de cuarenta cifras es primo. Nosotros saltaremos en esta jerarquía a un nivel relativamente bajo, pero desde el cual podamos subir y bajar.

Además, nos restringiremos a describir computadores con la llamada «arquitectura de Von Neumann». No nos preocupemos de la palabra

«arquitectura», que es simplemente una palabra larga para describir cómo se ordenan las cosas, estamos ordenando componentes electrónicos en vez de ladrillos y columnas. Von Neumann fue un matemático famoso que, además de hacer importantes contribuciones a los fundamentos de la mecánica cuántica, fue el primero en establecer claramente los principios básicos de los computadores modernos.[1] También tendremos ocasión de examinar el comportamiento de diferentes computadores trabajando sobre el mismo problema y cuando lo hagamos, nos ceñiremos a computadores que trabajan en serie en vez de en paralelo; esto es, aquellos que se turnan para resolver las diferentes partes del problema en lugar de trabajar simultáneamente. Todo lo que nos vamos a perder por la omisión del «procesado paralelo» es simplemente velocidad, nada fundamental.

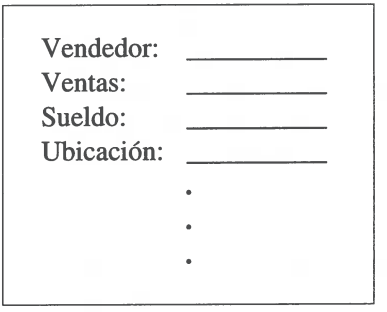
Hemos mencionado antes que la ciencia de la computación no es una verdadera ciencia. ¡Ahora vamos a renegar también de la palabra «computador»! La palabra «computador» nos hace pensar en la aritmética —suma, resta, multiplicación, y así sucesivamente— y es fácil suponer que esto es todo lo que un computador hace. De hecho, los computadores convencionales generalmente tienen un lugar dedicado a realizar las operaciones matemáticas básicas y el resto de la máquina se dedica a lo que constituye la tarea principal del computador, que es trasladar de un lado a otro trozos de papel —sólo que en este caso, las notas de papel son señales eléctricas digitales—. En muchos aspectos una computadora recuerda a un grupo de archiveros yendo y viniendo rápidamente a archivadores, sacando informes y volviendo a colocarlos, haciendo anotaciones en pequeños trozos de papel y pasando las notas a otros, y así sucesivamente; esta metáfora de papeleo en una oficina será un buen punto de partida para introducir algunas de las ideas básicas de la estructura de un computador. Lo vamos a tratar con detalle y algún impaciente entre vosotros puede pensar que es incluso demasiado detallado, pero es un modelo perfecto para transmitir qué es lo esencial de la tarea del computador y, por lo tanto, vale la pena dedicarle algún tiempo.

1. El modelo del archivero

Supongamos una gran empresa que emplea a muchos vendedores. En un gran sistema de archivos, en algún lugar, se almacena una cantidad

[1]. De hecho, hay actualmente mucho interés en el diseño de máquinas que «no son de tipo Von Neumann» y de las que tratarán expertos invitados en el siguiente volumen. [Editores.]

enorme de información sobre estos vendedores, y todo está organizado por un empleado. Comenzamos con la idea de que el archivero sabe cómo obtener la información del archivo. Los datos están almacenados en fichas y cada una de las fichas contiene el nombre del vendedor, su ubicación, el número y el tipo de ventas que ha hecho, su salario, y cosas así.



Supongamos ahora que tenemos que encontrar la respuesta a una pregunta específica: «¿Cuáles son las ventas totales en California?». Bastante aburrida y sencilla, y por eso la he elegido. Se debe empezar con preguntas sencillas para poder entender las preguntas difíciles después. ¿Cómo encuentra nuestro empleado el número total de ventas en California? Ésta es una forma de hacerlo:

Saca una ficha
Si la «ubicación» dice *California*, entonces
    Suma el número que aparece en «ventas» a una suma parcial llamada «total»
Vuelve a archivar la ficha de «ventas»
Saca la siguiente ficha y repite.

Obviamente hay que repetir este proceso hasta que se han repasado todas las fichas. Supongamos, ahora, que hemos sido tan desafortunados como para contratar a archiveros especialmente estúpidos que pueden leer, pero que encuentran las instrucciones anteriores demasiado difíciles de seguir, digamos que no saben llevar una suma parcial. Tendremos que ayudarles un poco más. Vamos a inventar una ficha llamada «total» para nuestro archivero que la utilizará para mantener la suma parcial de la siguiente manera:

Saca la siguiente ficha de «ventas»
Si *California,* entonces

Saca la ficha «total»
Suma el número de ventas al número en la ficha
Vuelve a archivar la ficha «total»
Archiva la ficha «ventas»
Saca la siguiente ficha «ventas» y repite.

Ésta es una representación muy mecánica de cómo resolvería este problema de sumación un computador rudimentario. Obviamente, los datos no tienen que estar almacenados en fichas y la máquina no tendría que «sacar una ficha»: podría leer la información almacenada en un registro. También podría escribir de un registro a una «ficha» sin necesidad de volver a archivarla físicamente.

¡Vamos a exigir un poco más a nuestro archivero! Supongamos que cada vendedor recibe además de su salario una pequeña comisión por ventas. Para calcular la comisión tenemos que multiplicar sus ventas por un porcentaje apropiado. Queremos encargar a nuestro archivero que haga este cálculo, pero él, que cobra poco y es muy rápido, desgraciadamente es demasiado torpe para multiplicar.[2] Si le decimos que multiplique 5 por 7 diría «¿qué?», así que tendremos que enseñarle a multiplicar. Para hacerlo, nos aprovecharemos del hecho de que hay algo que hace muy bien: puede sacar las fichas muy, muy rápidamente.

Trabajaremos en base dos. Como probablemente sabéis todos, las reglas de la aritmética binaria son más sencillas que en base diez; la tabla de multiplicar es tan pequeña que cabe fácilmente en una ficha. Supondremos, incluso, que nuestro archivero puede recordarlas; todo lo que necesita hacer son las operaciones «desplazar» y «llevar» como se explica en el siguiente ejemplo:

En decimal: $22 \times 5 = 110$

En binario:

```
   10110          en decimal: 22
     101                      5
  ------
   10110
10110   (desplazar dos veces)
-------
1101110                   110
```

[2]. Un inciso, aunque en estos ejemplos se supone que nuestro torpe archivero es un hombre, ¡no se pretende ninguna implicación sexista! [RPF]

Por lo tanto, mientras nuestro archivero pueda desplazar y llevarse, puede efectivamente multiplicar. Lo hace muy tontamente, pero lo hace con mucha rapidez y ésa es la razón de todo esto: ¡el interior de un computador es así de estúpido, pero va como loco! Puede realizar muchos millones de operaciones sencillas en un segundo y es como un archivero estúpido pero muy rápido. Precisamente por hacer las cosas tan deprisa, no nos damos cuenta de que las hace tan estúpidamente. (Es bastante interesante recordar que las neuronas del cerebro emplean milisegundos para realizar operaciones elementales, lo que nos deja con el siguiente misterio: ¿Por qué es el cerebro tan inteligente? Las computadoras pueden ser capaces de dejar parado al cerebro cuando multiplican, pero tienen problemas con cosas que incluso los niños pequeños hacen fácilmente, tales como reconocer a las personas o manipular objetos.)

Para seguir avanzando necesitamos especificar con más precisión nuestro conjunto básico de operaciones. Una de las operaciones más elementales es la transferencia de información de las fichas que lee nuestro archivero a un tipo de cuaderno de notas en el cual hace las cuentas:

OPERACIONES DE TRANSFERENCIA

«Saca la ficha X» = información de la ficha X escrita en el cuaderno
«Reemplaza la ficha Y» = información en el cuaderno escrita en la ficha Y.

Todo lo que hemos hecho ha sido definir la instrucción «sacar la ficha X» como copiar la información contenida en la ficha X al cuaderno, e igualmente con «reemplazar la ficha Y». El siguiente paso es enseñar al archivero cómo comprobar si la ubicación en la ficha X es «California». Él tiene que hacer esta comprobación para cada ficha, de manera que la primera tarea es recordar la palabra «California» de una ficha a la siguiente. Una forma de ayudarle a hacer esto es escribir *California* en otra ficha, *C*, de manera que las instrucciones ahora son:

Saca la ficha X (del archivador al cuaderno)
Saca la ficha C (del archivador al cuaderno)
Compara lo que hay en la ficha X con lo que hay en la ficha C.

Luego le decimos que si los contenidos coinciden tiene que hacer tal y tal cosa, y que si no es así vuelva a colocar las fichas en su sitio y saque las siguientes. Pero estar guardando y sacando la ficha de California

parece poco eficiente, y, de hecho, no hay que hacerlo; se puede mantener en el cuaderno mientras tanto. Esto sería desde luego mejor, pero depende de cuanto espacio tenga el archivero en su cuaderno y cuanta información necesite mantener. Si no hay mucho espacio tendrá que haber mucho trasiego de fichas de aquí para allá. ¡Tendremos que preocuparnos de cosas como ésas!

Podemos seguir descomponiendo las complicadas tareas del archivero en otras más sencillas y fundamentales. ¿Cómo, por ejemplo, conseguimos que mire a la parte «ubicación» de la ficha? Una forma sería cargar al pobre hombre con otra ficha en la cual estuviera escrito algo como esto:

0000 0000 0000 0000 0000 1111 0000 0000 0000 000...

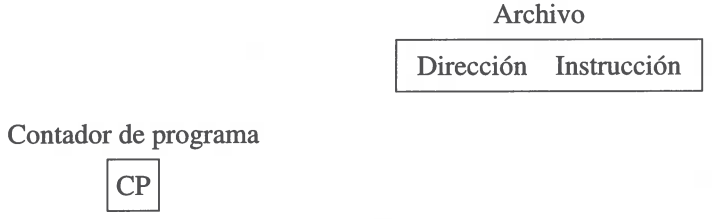
Cada secuencia de dígitos está asociada con un trozo particular de información en la ficha: el primer conjunto de ceros está «alineado» con el nombre del vendedor, el siguiente con su edad, y así sucesivamente. El archivero comprueba la lista numérica hasta que llega a un conjunto de unos y entonces lee la información que sigue a continuación. En nuestro caso la línea de 1111 está alineada con California. Este tipo de procedimiento de localización es realmente utilizado en los computadores, donde se emplea la llamada operación «AND bit a bit» (la veremos más adelante). He hecho esta pequeña digresión para llamar la atención sobre el hecho de que no necesitamos contar con ninguna de las habilidades del archivero: podemos conseguir que haga las cosas cada vez más estúpidamente.

2. Conjuntos de instrucciones

Echemos un vistazo al cuaderno de notas del archivero. Aún no le hemos enseñado cómo utilizarlo, así que lo haremos ahora. Supondremos que las instrucciones que lleva a cabo se pueden dividir en dos grupos. En primer lugar, hay un «conjunto básico de instrucciones» que comprende los procedimientos sencillos que vienen con el cuaderno, tales como sumar, transferir, etc. Están en el hardware y, por lo tanto, no se modifican cuando cambiamos de problema. Si se prefiere, reflejan las habilidades básicas del archivero. Luego tenemos un conjunto que es específico de la tarea en cuestión, digamos calcular las comisiones de los vendedores. Los elementos de este conjunto se construyen a partir de las

instrucciones del conjunto básico de la forma que hemos visto y representan combinaciones de las habilidades del archivero que serán necesarias para realizar la tarea que tiene encargada.

Lo primero que tenemos que conseguir es que el archivero haga las cosas en el orden adecuado, es decir, que siga una sucesión de instrucciones. Esto lo hacemos asignando una de las zonas de almacenamiento del cuaderno a un «contador de programa». Tendrá un número que indica en qué lugar del procedimiento de cálculo se encuentra el archivero. En lo que concierne al archivero el número es una dirección, él sabe que en medio del sistema de archivadores hay un cajón especial con el «archivo de instrucciones» y que el número en el contador selecciona la ficha en ese archivo que tiene que extraer; en esa ficha está la instrucción de lo que tiene que hacer a continuación. Así consigue la siguiente instrucción que apunta en su cuaderno en la zona que llamamos «registro de instrucción».



Sin embargo, antes de realizar la instrucción se prepara para la siguiente, incrementando el contador de programa, simplemente añadiéndole un 1. A continuación hace lo que quiera que la instrucción en el registro le dice que le haga. Utilizando una notación con paréntesis donde () significa «contenido de» —recordad esto porque lo utilizaremos muy a menudo— podemos escribir esta secuencia de acciones de la siguiente forma:[3]

Buscar instrucción en la dirección CP
CP ← (CP) + 1
Ejecutar la instrucción.

La segunda línea es una forma especial de decir que el contador *CP* «toma» el nuevo valor (*CP*) + 1. El archivero también necesitará algunas

[3]. Las convenciones adoptadas para este «lenguaje de transferencia de registro» cambian dependiendo del autor. Aquí hemos decidido seguir la llamada convención «derecha a izquierda» utilizada en los lenguajes de programación estándar. [Editores.]

áreas de almacenamiento temporal en el cuaderno donde pueda, por ejemplo, hacer las cuentas. Estas áreas se llaman registros y son lugares para almacenar algún dato mientras se va a buscar algún otro número. ¡Incluso si sólo se trata de sumar dos números se necesita recordar el primero hasta que se ha obtenido el segundo! Todo debe ser secuencial y los registros nos permiten así organizar las cosas. Generalmente tienen nombres; en nuestro caso, tendremos cuatro registros que llamaremos *A*, *B*, *X* y el cuarto, *C*, que es especial porque sólo puede almacenar un bit de datos y nos referiremos a él como el registro «transportador». Podríamos tener más o menos registros. En general, cuantos más registros se tengan más fácil es escribir el programa, pero cuatro serán suficientes para nuestro propósito.

Así, nuestro archivero ahora sabe lo que tiene que hacer y cuándo hacerlo. Veamos el conjunto de instrucciones básicas para este cuaderno. El primer tipo de instrucción está relacionado con la transferencia de datos de una ficha a otra. Por ejemplo, supongamos que tenemos una posición de memoria, *M*, sobre el cuaderno y deseamos una instrucción que transfiera el contenido del registro *A* a *M*:

Transferir (A) a M o M ← (A).

De igual manera, quizá queramos hacerlo al revés y escribir el contenido de *M* en *A*:

Transferir (M) a A o A ← (M).

Por cierto, que *M* no tiene que ser necesariamente un almacenamiento temporal como ocurre con *A*. También debemos tener instrucciones análogas para el registro *B*:

Transferir (B) a M o M ← (B)
Transferir (M) a B o B ← (M).

El registro *X* se utilizará de una manera un poco diferente. Permitiremos transferencias de *B* a *X* y de *X* a *B*:

X ← (B) y B ← (X).

Además, necesitaremos controlar y manipular nuestro contador de programa *CP*. Esto es obviamente necesario: si, por ejemplo, el archivero se dedica a ejecutar alguna multiplicación, cuando termina y regresa

tiene que saber qué hacer a continuación, tiene que recordar el número que hay en *CP*. De hecho, lo mantendremos en el registro *X*. Así, añadiremos las instrucciones de transferencia:

$$\text{CP} \leftarrow (\text{X}) \quad \text{y} \quad \text{X} \leftarrow (\text{CP}).$$

A continuación, necesitaremos algunas operaciones aritméticas y lógicas. La más básica de éstas es una instrucción «borrar»:

$$\text{Borrar A}, \quad \text{o} \quad \text{A} \leftarrow 0.$$

Esto significa que haya lo que haya en *A*, se debe borrar. Luego necesitaremos la operación suma:

$$\text{Suma B a A}, \quad \text{o} \quad \text{A} \leftarrow (\text{A})+(\text{B}).$$

Esto significa que el registro *A* recibe la suma del contenido de *B* y el contenido previo de *A*. También tenemos una operación de desplazamiento, que nos permitirá hacer multiplicaciones sin tener que introducir una instrucción básica para ello:

$$\text{desplazar } A \text{ a la izquierda} \quad \text{y} \quad \text{desplazar } A \text{ a la derecha}.$$

La primera simplemente desplaza todos los bits de *A* una posición a la izquierda. Si este desplazamiento hace que el bit situado más a la izquierda desaparezca lo almacenamos en el registro de acarreo *C*. También podemos desplazar nuestro número hacia la derecha; ahora mismo no se me ocurre para qué, ¡pero podría ser útil!

Las siguientes instrucciones son lógicas. Aunque las veremos con más detalle en el siguiente capítulo, las mencionaré aquí por completitud. Hay tres que nos van a interesar: AND, OR y XOR. Cada una de las cuales es una función de dos «entradas» digitales *x* e *y*. Si *ambas* entradas son 1, el resultado de AND es 1, en cualquier otro caso es 0. Como veremos, la operación AND aparece en la suma binaria y, por lo tanto, en la multiplicación; si vemos *x* e *y* como dos números que estamos sumando entonces (*x* AND *y*) es el bit de acarreo, que es 1 solamente si ambos dígitos son 1. Utilizando nuestros registros, *x* e *y* son (*A*) y (*B*), y AND opera sobre éstos de la forma:

$$\text{AND: A} \leftarrow (\text{A}) \wedge (\text{B}),$$

donde hemos utilizado el símbolo lógico ∧ para designar la operación AND. El resultado de aplicar un operador tal como AND sobre un par de variables a menudo se resume en una «tabla booleana» (Tabla 1.1):

TABLA 1.1 La tabla booleana para el operador AND.

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

$X = A \wedge B$

De igual forma podemos describir los otros dos operadores. OR también opera sobre (*A*) y (*B*), el resultado es 1 a menos que tanto (*A*) como (*B*) sean 0 —(*x* OR *y*) es 1 si *x* o *y* son 1. El operador XOR, o «OR exclusivo» es similar a OR, excepto en que da 0 si (*A*) y (*B*) son 1. En la suma binaria de *x* e *y* corresponde a lo que se obtiene si se suman *x* e *y*, ignorando el bit de acarreo. La suma binaria de 1 y 1 es 10, que es 0 si nos olvidamos del 1 que nos llevamos. Podemos introducir los siguientes símbolos lógicos relevantes:

$$\text{OR} \quad A \leftarrow (A) \vee (B)$$
$$\text{XOR} \quad A \leftarrow (A) \oplus (B)$$

Los efectos de aplicar OR y XOR se pueden resumir en las siguientes tablas booleanas:

TABLA 1.2 Tablas booleanas de los operadores OR y XOR.

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

OR

$X = A \vee B$

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

XOR

$X = A \oplus B$

Otras dos operaciones que resulta conveniente tener en cuenta son las instrucciones para incrementar y disminuir en una unidad el contenido de *A*:

Incrementa A, o A ← (A) + 1
Disminuye A, o A ← (A) – 1

Obviamente, uno podría seguir añadiendo instrucciones que pueden o no pueden ser muy convenientes. Ya tenemos más del mínimo necesario para hacer algunos cálculos útiles. Sin embargo, como queremos hacer tanto como sea posible vamos a introducir unas instrucciones más. Una que podría ser útil es la que nos permite introducir datos directamente en un registro. Por ejemplo, en vez de escribir *California* en una ficha y luego tener que transferir el dato desde la ficha al cuaderno, sería conveniente poder escribir *California* directamente en el cuaderno. Así, introducimos la instrucción «carga directa»:

Carga directa: B ← N,

donde *N* es cualquier constante.

Hay un tipo de instrucciones que es vital añadir: las ramificaciones o saltos. Un «salto a *Z*» es básicamente una orden al archivero de que mire en la ubicación *Z*; esto es, implica un cambio en el valor del contador del programa mayor que el incremento usual de una unidad. Esto permite a nuestro archivero saltar de una parte a otra del programa. Hay dos clases de saltos, el «incondicional» y el «condicional». El salto incondicional ya lo hemos visto antes:

Salta a (Z) o CP ← (Z).

Lo que es realmente nuevo es el salto condicional:

Salta a (Z) *si* C = 1.

Con esta instrucción el salto a la ubicación (*Z*) sólo se realiza si el registro de acarreo *C* contiene el bit 1. La libertad que nos proporciona esta instrucción condicional es vital para el diseño de cualquier máquina interesante.

Hay otros muchos tipos de saltos que podemos añadir. Algunas veces es conveniente poder saltar no solamente a un lugar definido sino también un número específico de pasos sucesivos en el programa. Pode-

mos introducir, por lo tanto, instrucciones de salto que añadan este número de pasos al contador del programa

$$\text{Salta a (CP)} + \text{(Z)} \quad \text{o} \quad \text{CP} \leftarrow \text{(CP)} + \text{(Z)},$$
$$\text{Salta a (CP)} + \text{(Z) si C} = 1.$$

Finalmente, hay otra orden más que vamos a necesitar, es la instrucción que le dice a nuestro archivero que termine:

Parar.

Con estas instrucciones podemos ahora hacer todo lo que queramos, y un poco más adelante os sugeriré algunos problemas para que practiquéis. Antes de hacer eso dejadme resumir dónde estamos y qué estamos tratando de hacer. La idea ha sido describir esquemáticamente las operaciones y los métodos básicos de un computador e indicar lo que realmente tiene dentro (no he estado describiendo un diseño real, pero me he acercado bastante). En un computador sencillo hay solamente unos pocos registros y aunque los más complejos tienen más, los conceptos son básicamente los mismos, simplemente un poco agrandados.

Vale la pena mirar cómo representamos las instrucciones que hemos considerado anteriormente. En nuestro caso particular, las instrucciones contienen dos partes: una dirección de instrucción y un número de instrucción:

Dirección de la instrucción	número de la instrucción

Por ejemplo, una de las instrucciones era: «poner el contenido de la memoria *M* en el registro *A*». El computador no habla nuestro idioma de manera que tendremos que codificar esta orden en una forma que la pueda entender; en otras palabras, en un número binario. Éste es el número de instrucción, y su longitud determina claramente el número de instrucciones diferentes que podemos tener. Si el número de instrucción es un número binario de cuatro dígitos podemos tener $2^4 = 16$ instrucciones diferentes, una de las cuales corresponde a cargar el contenido de una dirección de memoria en *A*. La segunda parte de la instrucción es la dirección de la instrucción, que le dice al computador a dónde tiene que ir para encontrar lo que tiene

que cargar en A, esto es, la dirección de memoria *M*. Algunas instrucciones tales como «borrar *A*» no requieren dirección alguna.

Para utilizar las instrucciones no es necesario conocer los detalles de cómo se representan los números de instrucción de las instrucciones o cómo se colocan las cosas en la memoria. Éste es el primer nivel y el más elemental, en una serie de jerarquías. Queremos ser capaces de mantener tal ignorancia consistentemente. Dicho en otras palabras, queremos tener que pensar en los detalles inferiores una sola vez y diseñar las cosas de tal manera que el siguiente que venga y desee utilizar su estructura no tenga que preocuparse de los detalles de ese nivel inferior.

Ésta es una característica que hemos ignorado completamente hasta ahora. Nuestra máquina, tal y como la hemos descrito, no podría funcionar porque no tenemos una forma de introducir y obtener números. Tenemos, pues, que considerar la entrada y la salida. Una forma rápida de resolver este asunto consistiría en asignar un lugar determinado de la memoria, digamos la dirección 17642, a la entrada y asociarla al teclado de manera que cualquiera desde el exterior de la máquina pueda cambiar su contenido. De la misma forma, otra posición, digamos la 17644, podría ser la salida, que estaría conectada a un monitor de TV o a algún otro dispositivo, de manera que los resultados de un cálculo pudieran salir al mundo exterior.

Hay dos maneras para profundizar en la comprensión de estos temas. Una forma es memorizar las ideas generales, irte a casa, tratar de deducir los comandos que necesites, asegurándote de no olvidar ninguno, ampliar o recortar este conjunto según convenga y evaluar tu elección mediante la resolución de problemas. Ésta es la manera en que yo lo haría porque ¡ésta es mi forma de ser! Es la forma en la que estudio —entender algo intentando resolverlo o, en otras palabras, entender algo creándolo—. No creándolo al cien por cien, desde luego, pero sí a partir de una idea de la dirección en que hay que ir, pero sin recordar los detalles. Ésos los obtiene uno mismo.

La otra forma, que también es valiosa, es leer cuidadosamente cómo lo ha hecho alguien antes. Para mí, una vez que he entendido la idea básica, el primer método es mejor. Si me quedo atascado miro en un libro que me diga cómo lo hizo algún otro. Voy leyendo las páginas hasta que digo «ah, olvidé ese detalle» y, entonces, cierro el libro y continúo. Al final, cuando he resuelto cómo resolverlo, leo cómo lo han hecho otros y descubro lo torpe que es mi solución y lo eficiente e inteligente que es la de los demás. De esta forma puedo entender la brillantez de sus ideas y adquirir un marco en el que pensar sobre el problema. Cuando comienzo directamente a leer la solución de alguien lo encuentro aburrido, poco in-

teresante y no hay manera de ver el problema en su conjunto.¡Al menos, ésa es la forma que funciona para mí!

A lo largo del libro, os sugeriré algunos problemas para que practiquéis con ellos. Podéis sentir la tentación de saltárlos. Si son demasiado difíciles, bien. ¡Algunos son bastante difíciles! Pero también podríais saltaros algunos pensando que, bien, probablemente ya los haya resuelto alguien, de manera que ¿para qué voy a hacerlos yo? ¡Desde luego que ya se han resuelto! Pero ¿y qué? Resolvedlos por el *placer* de hacerlos. Así es como se aprenden los trucos para poder hacer las cosas cuando llegue el momento. Os daré un ejemplo. Supongamos que quiero sumar la siguiente serie de números:

$$1 + 2 + 3 + 4 + 5 + 6 + 7...$$

hasta, digamos, 62. Sin duda, sabéis cómo hacerlo, pero cuando de niños jugáis con este tipo de problemas y no os han enseñado la respuesta... resulta *divertido* tratar de imaginarse cómo hacerlo. Luego, al entrar en la edad adulta, desarrolláis cierta confianza en que podéis descubrir cosas por vosotros mismos; pero que otros lo hayan hecho antes no debiera importaros en absoluto. Lo que un necio hace, otro puede también hacerlo, y el que un necio se os adelante no tendría por qué quitaros el sueño: contento deberíais estar por haber descubierto algo. La mayoría de los problemas que voy a proponer en este libro han sido resueltos numerosas veces y muchas de las soluciones son muy ingeniosas. Pero si seguís probando cosas que otros ya han hecho, adquiriendo confianza, aumentando la complejidad de vuestras soluciones —por el placer de ello— entonces, un día descubriréis que *¡realmente nadie ha hecho esto!* Y ésa es la manera de llegar a ser un científico de la computación.

Os daré un ejemplo de esto obtenido de mi propia experiencia. Antes he mencionado la suma de enteros. Hace muchos años me interesé por la generalización de ese problema: quería obtener fórmulas para la suma de los cuadrados, cubos y potencias superiores, tratando de encontrar la suma de m números cada uno elevado a la potencia n. Y lo conseguí, encontré un montón de bonitas relaciones. Cuando había acabado tenía una fórmula para cada suma en función de un número, distinto para cada valor de n, pero para el cual no pude encontrar una fórmula. Fui obteniendo esos números pero no fui capaz de encontrar una regla general para calcularlos. Lo interesante es que eran números enteros hasta llegar a $n = 13$, momento en que dejaban de serlo (era algo dividido por 691). ¡Muy chocante y divertido!

En cualquier caso, más tarde me enteré que estos números habían sido descubiertos allá por 1746. ¡Así que había llegado hasta 1746! Se llaman «números de Bernoulli». La fórmula para obtenerlos es bastante complicada y no se conoce una forma sencilla. Yo tenía una «relación de recurrencia» que me permitía obtener el siguiente a partir del anterior, pero no conseguí hallar la expresión de uno arbitrario. Así discurrió mi vida, descubriendo más tarde algo que ya había sido encontrado en 1889, después algo de 1921... y finalmente hallando algo que llevaba la misma fecha que cuando yo lo descubrí. Pero me lo pasé tan bien haciendo esto que me imagino que debe haber otras personas por ahí haciendo lo mismo. Por eso os voy a proponer estos problemas para que los disfrutéis. (Desde luego, cada uno disfruta de forma diferente.) Yo os animo a que no os sintáis intimidados por ellos y que no los abandonéis por el simple hecho de que hayan sido ya resueltos. Es poco probable que descubráis algo nuevo sin practicar mucho sobre viejas cuestiones, y además disfrutaréis deduciendo relaciones curiosas y cosas interesantes. Además, si luego leéis lo que hizo algún otro loco, podréis apreciar lo difícil que fue hacerlo (o no), lo que estaba tratando de hacer, cuáles eran sus problemas, y así sucesivamente. Es mucho más fácil entender las cosas después de haber estado enredando con ellas antes de leer la solución. Así que, por todas estas razones, os sugiero que lo intentéis.

Problema 1.1: *a*) Volvamos a nuestro estúpido archivero y al problema de encontrar el total de ventas en California. ¿Aconsejaríais a la dirección contratar dos archiveros para hacer el trabajo más deprisa? Si es así, ¿cómo los utilizaría y cómo podría aumentar la velocidad del cálculo en un factor 2? Tenéis que pensar cómo obtienen los archiveros sus instrucciones. ¿Podéis generalizar vuestra solución para K o incluso para 2^K archiveros?

b) ¿Qué clase de problemas pueden K archiveros realizar realmente más deprisa? ¿Qué tipo de problemas parece que no pueden ser acelerados?

c) La mayoría de los computadores actuales tienen solamente un procesador central —para utilizar nuestra analogía, un archivero—. Este único archivero está sentado todo el día trabajando con entusiasmo, sacando y metiendo fichas del archivador sin parar. Al final, la velocidad de toda la máquina está determinada por la velocidad con la cual el archivero, esto es, el procesador central, puede hacer estas operaciones. Veamos cómo se puede mejorar el rendimiento de la máquina. Supongamos que queremos comparar dos números de n bits, donde n es un número

grande, tal como 1024; queremos ver si los dos números son iguales. La forma más sencilla para un único archivero sería comparar todos los números de la secuencia. Obviamente, esto le llevará un tiempo proporcional a n, el número de dígitos que hay que comparar. Pero supongamos que podemos contratar n archiveros o $2n$ o quizá $3n$, depende de nosotros decidir cuantos, con tal que el número sea proporcional a n. En este caso, resulta que aumentando el número de archiveros podemos obtener que los tiempos disminuyan de forma proporcional al $\log_2 n$. ¿Podéis ver cómo?

d) Si podéis resolver este problema de comparación, podríais intentar uno más difícil. Ved si podéis descubrir la forma de sumar dos números de n bits en un tiempo «log n». ¡Esto es más difícil porque tienen que ocuparse de las llevadas o acarreos!

Problema 1.2: El segundo problema consiste en conseguir que el archivero multiplique (recordad que la multiplicación no está incluida en su juego de instrucciones básicas). El problema tiene dos partes. Primero, encontrar el juego apropiado de instrucciones básicas necesario para realizar la multiplicación. Obtenido éste, supongamos que lo guardamos en algún lugar de la máquina de manera que no tengamos que duplicarlo cada vez que deseamos multiplicar; ponedlo, digamos, en las posiciones m a $m + k$. Encontrar la manera de darle al archivero las instrucciones para que utilizando este conjunto haga una multiplicación y pueda volver al lugar adecuado en el programa.

3. Resumen

Ya hemos visto suficiente materia como para entender cualquier diseño particular de una máquina. Pero en vez de mirar con detalle una máquina concreta vamos a hacer algo bastante diferente. Desde donde estamos podemos ir hacia arriba, hacia abajo o lateralmente. ¿Qué quiero decir con esto? Bien, «arriba» significa ocultarle al usuario más detalles del funcionamiento de la máquina —introducir más niveles de abstracción—. Ya hemos visto algunos ejemplos de esto; por ejemplo, construir nuevas operaciones tales como la multiplicación a partir de operaciones de nuestro juego básico. Cada vez que deseamos multiplicar, simplemente utilizamos esta «subrutina» multiplicación. Otro ejemplo que vale la pena discutir es la capacidad para hablar de variables algebraicas en lugar de posiciones de memoria. Supongamos que queremos calcular la suma de X e Y, y llamar al resultado Z:

$$Z = X + Y$$

el computador sabe qué son *X* e *Y* y las tiene guardadas en posiciones específicas de la memoria. Lo primero que tenemos que hacer es asignar algún lugar de la memoria para almacenar el valor de *Z*, y luego asegurarnos de que esta posición contiene la suma de los contenidos de las unidades de memoria *X* e *Y*. Ahora sabemos todo acerca de *Z* y podemos utilizarla en otras instrucciones tales como en *Z* + *X*. Aunque sea una tarea complicada, generalmente es mucho más sencillo hablar de variables algebraicas que estar continuamente hablando de posiciones de memoria. Sin embargo, hasta ahora hemos tenido que saber exactamente dónde estaba un número almacenado para poder transferirlo. Ahora podemos introducir un nuevo número *Z* y decirle al computador «¡yo quiero un número *Z*, encuentra un sitio donde guardarlo y no me digas dónde!». Esto es lo que quiero decir con ir hacia «arriba».

Desde luego, ya hemos ido un poco hacia «arriba» cuando resumimos operaciones por instrucciones tales como «borra *A*» y otras así. Este tipo de taquigrafía se introduce para nuestra conveniencia, pero la máquina no puede entender directamente los programas así escritos. Estos programas escritos en «lenguaje ensamblador» tienen que ser traducidos a «lenguaje máquina» que el computador sí puede entender y esto se hace mediante un programa llamado «ensamblador». El siguiente nivel hacia arriba, donde tenemos la multiplicación, variables y cosas así, necesita otro programa que traduzca estos programas de «alto nivel» en lenguaje ensamblador. Estos programas de traducción se llaman «compiladores» o «intérpretes». La diferencia entre ambos está en cuándo se hace la traducción. Un intérprete traduce lo que tiene que hacer paso a paso, según avanza el programa, interpretando cada una de las instrucciones en términos de un lenguaje más elemental. El compilador coge el programa como un conjunto y lo convierte del todo en ensamblador o lenguaje máquina, antes de que el programa empiece a ejecutarse. Los compiladores tienen la ventaja de que en algunos casos, mirando el «código» en su conjunto son capaces de encontrar caminos más eficientes para realizar las operaciones requeridas. Ésta es la esencia de un campo tan importante como el de la «optimización del compilador» que adquiere una relevancia cada vez mayor para los nuevos computadores paralelos de tipo «no von Neumann».

Está claro que uno puede continuar subiendo niveles, juntando nuevos algoritmos y lenguajes de programación, añadiendo facilidades para manipular «archivos» que contienen programas y datos, y así sucesiva-

mente. Actualmente la mayoría de las personas pueden trabajar felizmente en estos niveles utilizando lenguajes de alto nivel para programar sus máquinas. ¡Imagínese lo tedioso que era, y aún lo es para los diseñadores de computadores, trabajar únicamente en código máquina!

Esto fue hacia «arriba», ahora ha llegado el tiempo de ir hacia «abajo». ¿Cómo puede ser algo más simple que nuestro modelo del archivero estúpido y nuestra sencilla lista de instrucciones? Lo que aún no hemos visto es de qué *está hecho* nuestro archivero; para ser más realistas, no hemos visto cómo los circuitos electrónicos realizan las diversas operaciones que hemos discutido. Ahí es adonde vamos a ir a continuación; pero antes de hacerlo, permitidme explicar lo que quiero decir con moverse «lateralmente». Lateralmente significa mirar algo completamente diferente a nuestra arquitectura von Neumann, que está caracterizada por tener una única unidad de procesador central («CPU») y todo entra y sale a través del ciclo «obtener y ejecutar». Se está experimentando con muchas otras arquitecturas exóticas de computador y algunas se han comercializado como máquinas que la gente puede comprar. Ir «lateralmente», por lo tanto, significa permanecer al mismo nivel de detalle, pero examinar cómo se podrían realizar los cálculos en máquinas con diferentes estructuras básicas. Ya os hemos invitado a pensar sobre computadores «paralelos» con el problema de la organización de varios archiveros trabajando juntos en el mismo problema.