



R621.381
E452.2
Y.8, v. 1

Enciclopedia de la **ELECTRONICA** **INGENIERIA Y TECNICA**

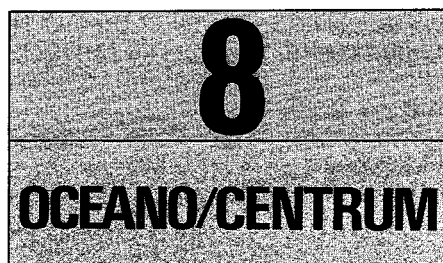
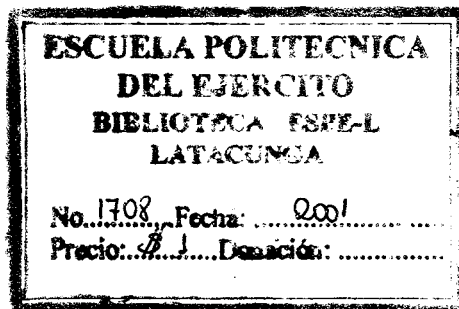
Charles Belove

**Departamento de Ingeniería Eléctrica y Computación
Florida Atlantic University
Boca Raton, Florida**

Director de la versión en español

Francisco Paniagua B., I.M.E.

Universidad Nacional Autónoma de México
Consultor Editorial de Ingeniería Eléctrica
Miembro de la U.S. Metric Association



Es una obra del
GRUPO EDITORIAL OCEANO

Presidente
José Lluís Monreal
Director General
José M.^a Martí
Director General de Publicaciones
Carlos Gisbert



Versión española de la edición original de Charles Belove "Handbook of Modern Electronics and Electrical Engineering" publicada por John Wiley & Sons, Nueva York, EE.UU.

VERSIÓN ORIGINAL

Consejo Editorial: Phillip Hopkins
Lockheed Engineering and
Management Services, Co., Inc.
Houston, Texas

Edward Nelson
New York Institute of Technology
Old Westbury, Nueva York

Milton Rosenstein
New York Institute of Technology
Old Westbury, Nueva York

Stanley Shinnars
Jericho, Nueva York

VERSIÓN ESPAÑOLA

Supervisión Editorial: Pedro Basurto Samperio

Edición: Begoña Robles

Dirección Editorial: Marta Bueno

Traducción: Roberto Palacios Martínez
Licenciado en Ciencias
Universidad Autónoma de Baja
California

Hugo Villagómez Velázquez
Licenciado en Física y Matemáticas
Instituto Politécnico Nacional, México
Doctor en Ciencias
Universidad de París, Francia

Juan Carlos Vega Fagoaga
Ingeniería en Sistemas

José Rafael Blengio Pinto
Médico Cirujano
Universidad Nacional Autónoma
de México

M.^a Dolores García Díaz
Traductora Especializada en Ciencias

© MCMLXXXVI Edición Original John Wiley & Sons, Inc.

© MCMXC Edición Española Ediciones Centrum Técnicas y Científicas

Paseo de Gracia, 26 — 08007 Barcelona — España

Tel. (93) 301 01 82 — Télex 51 735 exit e — Fax (93) 317 97 01

Reservados todos los derechos. Quedan rigurosamente prohibidas, sin la autorización escrita de los titulares del copyright, bajo las sanciones establecidas en las leyes, la reproducción total o parcial de esta obra por cualquier medio o procedimiento, comprendidos la reprografía y el tratamiento informático, y la distribución de ejemplares de ella mediante alquiler o préstamo públicos.

ISBN Edición española obra completa: 84-7841-016-3

ISBN Edición española volumen 8: 84-7841-024-4

ISBN Edición original: 0-471-09754-3

Impreso en España — Printed in Spain

Depósito legal B: 2007-91 (En)

Imprime: HUROPÉ, S.A.

Recaredo, 2-4 Barcelona

Índice de capítulos

DECIMOSEGUNDA PARTE (CONTINUACIÓN) COMPUTADORAS

58. Programación, 1897

Peter G. Anderson, John A. Biles,
James R. Carbin, Warren R.
Carithers, James A. Chmura, Chris
Comte, Lawrence A. Coon, Mary
Ann Dvonch, Henry A. Etlinger,
James Hammerton, Jack
Hollingsworth, Guy Johnson, Peter H.
Lutz y Rayno D. Niemi

Programación estructurada, 1898
Panorama general del proceso
de programación, 1898
Software de sistemas, 1901
Lenguajes de programación, 1902
Simulación de sistemas, 1918
Sistemas de bases de datos, 1920

59. Organización del procesador central, 1923

Melvyn M. Drossman

Estructura y función, 1924
Unidad de control, 1925
Instrucciones, 1926
Organización de ductos, 1928
Operación de la CPU, 1928
Unidad aritmética y lógica, 1931
Algoritmos aritméticos, 1934
Microprogramación, 1944
Microprocesadores de "rebanadas"
de bits, 1951

60. Sistemas de memoria, 1953

Jay Michlin

Introducción, 1954
Definiciones, 1954
Hardware de memoria, 1956
Arquitecturas de memoria, 1957
Memoria virtual, 1961

61. Entrada y salida (I/O) de los sistemas de computación, 1965

Sam Goldwasser

Introducción, 1966
Tipos y ejemplos de dispositivos I/O,
1968
Consideraciones sobre sistemas
generales, 1976
Estructuras de ductos de sistemas I/O, 1978
Interfaz de hardware y software, 1982
I/O controlada por el programa, 1985
I/O activada por interrupción, 1986
I/O de acceso directo a la memoria, 1989
I/O mapeada por la memoria, 1990
Técnicas de I/O avanzadas, 1993
Resumen y tendencias en los sistemas
I/O, 1995

62. Microcomputadoras y microprocesadores, 1997

Edward J. Lancevich

Tipos y tecnologías de procesadores, 1998
Características arquitectónicas y modos
de direccionamiento, 1998

Conjuntos de instrucciones, 2006
Ejemplos de programación, 2013
Principios de vinculación, 2015
Dispositivos de memoria de microprocesadores, 2022
Controladores de dispositivos y circuitos de interfaz, 2024
Sistemas de desarrollo de microprocesadores y auxiliares de diseño, 2030
Selección del microprocesador, 2034

63. Ingeniería de software, 2037

Frederic L. Swern

Introducción, 2038
Metodología de la programación, 2038
Administración de proyectos de software, 2041

64. Graficado por computadora, 2047

Guy Johnson

Introducción, 2048
Hardware de graficado, 2048
Software de graficado, 2049
Conclusión, 2056

65. Redes de comunicación de computadoras, 2057

Richard Van Slyke

Elementos de redes, 2058
Topología de redes, 2060
Modos de conmutación, 2060
Proceso de diseño, 2062
Arquitectura y protocolos, 2064

**DECIMOTERCERA PARTE
ENERGÍA DE FUENTES ALTERNATIVAS**

66. Conversión de energía, 2069

Jerald D. Parker

Conversión termoeléctrica, 2070
Conversión termoiónica, 2072
Conversión fotovoltaica, 2073
Conversión por celdas de combustible, 2079
Conversión magnetohidrodinámica (MHD), 2082

Conversión de energía térmica solar, 2083
Conversión de energía eólica, 2089
Conversión geotérmica, 2094
Combustibles obtenidos de desechos, 2097
Conversión nucleoelectrica, 2100

67. Sistemas de energía eléctrica, 2105

Daniel D. Lingelbach

Sistemas de distribución, 2106
Fluctuación y regulación del voltaje, 2113
Corrección del factor de potencia y potencia reactiva (kVAr), 2117
Protección de sistemas, 2122
Dispositivos de protección contra sobrecorriente, 2128
Dispositivos de protección contra sobrevoltaje, 2130
Conexión a tierra del equipo y del sistema, 2132

68. Máquinas eléctricas e iluminación, 2135

Daniel D. Lingelbach

Tipos básicos de máquinas de CD y CA, 2136
Características de funcionamiento de las máquinas de CA, 2136
Características de funcionamiento de las máquinas de CD, 2139
Selección y aplicación de motores de CA y de CD, 2142
Fundamentos de control de motores, 2144
Iluminación, 2147

69. Administración de la energía, 2159

Wayne C. Turner

¿Qué es administración de la energía?, 2160
Diseño, inicio y operación de programas de administración de energía, 2160
Auditorías de energía, 2162
Listas de control para proyectos de administración de energía, 2165
Oportunidades en la administración de energía, 2170

Índice de materias, 2175

CAPÍTULO 58

Programación

Peter G. Anderson
John A. Biles
James R. Carbin
Warren R. Carithers
James A. Chmura
Chris Comte
Lawrence A. Coon
Mary Ann Dvonch
Henry A. Etlinger
James Hammerton
Jack Hollingsworth
Guy Johnson
Peter H. Lutz
Rayno D. Niemi

School of Computer Science and Technology
Rochester Institute of Technology
Rochester, Nueva York

58.1 Programación estructurada

58.2 Panorama general del proceso de programación

- 58.2.1 Escritura de un programa
- 58.2.2 Ejecución de un programa

58.3 Software de sistemas

58.4 Lenguajes de programación

58.4.1 Panorama general

58.4.2 Lenguajes específicos

58.5 Simulación de sistemas

- 58.5.1 GPSS
- 58.5.2 Simscript II.5
- 58.5.3 SIMULA
- 58.5.4 GASP
- 58.5.5 SLAM

58.6 Sistemas de bases de datos

58.1 PROGRAMACIÓN ESTRUCTURADA

"Programación estructurada" es un término que tiene diversos significados para diferentes personas. Si bien es difícil determinar el inicio exacto de la "revolución" de la programación estructurada, puede decirse que ésta se ha convertido en una disciplina. Esta disciplina incluye una serie de ideas que pueden aplicarse cuando la solución a un problema significa producir un programa para una computadora. El apego fiel a los principios de la programación estructurada lleva al desarrollo de un producto: un programa que posee ciertas características. Sin embargo, la programación estructurada pretende además esclarecer el proceso que se emplea para obtener el resultado deseado. El nombre de E. W. Dijkstra aparece una y otra vez cuando se hace un análisis retrospectivo de las raíces de la programación estructurada. En su obra *Programming Considered as a Human Activity*¹, Dijkstra propuso de manera convincente la división de problemas complejos en una serie de problemas de menor tamaño para su solución (el método de "divide y vencerás"). También hizo un recuento de sus experimentos con programas sin instrucciones GOTO, y describió la mayor claridad conseguida de esta forma y el flujo de control del programa en términos de instrucciones IF (en el caso de trayectorias de programa condicionales) e instrucciones WHILE (para trayectorias repetitivas; es decir, ciclos).

La base teórica de la programación estructurada fue dispuesta primero por Bohm y Jacopini y después por Ashcroft y Manna; ellos demostraron¹ la posibilidad de convertir cualquier diagrama de flujo en uno equivalente en el que sólo se utilizaran combinaciones de componentes estándares de diagramas de flujo "estructurados" (que corresponden a instrucciones IF y WHILE).

El documento de Dijkstra, "Structured Programming"¹, apoyó los programas "libres de instrucciones GOTO" principalmente para que se prestaran a pruebas de corrección (es decir, los programadores podrían razonar sus programas).

Desde mediados de la década de 1970 hasta el presente, la programación estructurada se ha incorporado en una metodología estructurada mayor, que comprende todos los aspectos de la producción de un programa. Esta nueva apreciación de la programación representa un adelanto creciente, hecho posible por quienes trabajan con computadoras.

La revolución de la programación estructurada ha traído consigo una riqueza de herramientas de expresión con las cuales deben trabajar los programadores. Un método particularmente útil es el del "seudocódigo" o "lenguaje de diseño del programa", en el cual las estructuras de control de la programación estructurada (instrucciones IF y WHILE y llamadas a procedimientos) se combinan con un texto ordinario en inglés para describir un programa al más alto nivel (es decir, menos detallado). Este método, que sustituye al clásico diagrama de flujo como técnica de diseño y documentación, se aprovecha en la sección que sigue, en que se analiza un programa ordenador de burbuja.

58.2 PANORAMA GENERAL DEL PROCESO DE PROGRAMACIÓN

58.2.1 Escritura de un programa

En esta sección se presenta un panorama general de la forma en que se genera un programa. Para ayudar en la exposición se escribirá un programa pequeño mediante el uso de un lenguaje conocido como seudocódigo o lenguaje de diseño de programas (PDL), que es una mezcla informalmente definida de un texto ordinario en inglés y sintaxis parecida a la del lenguaje Pascal o Ada. Las ventajas del seudocódigo radican en que puede ser más fácil de entender que los lenguajes de programación formales y no abruma al programador con detalles de sintaxis que no tienen importancia durante el diseño del algoritmo. Sin embargo, como el seudocódigo es similar al lenguaje de la implementación final, la primera aproximación a una solución puede evolucionar en forma natural hacia un producto terminado en forma gradual y ordenada. Las estructuras básicas de control de seudocódigo se ilustran en la figura 58-1.

El programa por escribir ordenará una lista de N enteros en orden de magnitud creciente. El método que se aplicará consiste en buscar el valor más grande en la lista de entrada, desplazar al final de la lista y repetir el proceso para los $N - 1$ valores restantes. El proceso continúa hasta que sólo falte un valor por ser ordenado (una lista de longitud 1, por definición, está ordenada). Este método de desplazar el elemento más grande al final de la lista servirá para comparar elementos adyacentes, intercambiándolos si están fuera de orden. Este método se denomina ordenamiento de burbuja debido a que el elemento más pequeño se eleva como una burbuja hasta la parte más alta de la lista².

La lista de números se representa por medio de un nombre, y los elementos individuales de la lista se representan por medio del nombre de la lista y un subíndice. Según esta norma, el primer elemento de la lista A se denota A[1], y el elemento 21 de la lista B se denota B[21].

En el caso del programa que aquí se considera, se llamará LIST a la lista de números, y se declara la intención de utilizar la lista de la figura 58-2. Esta declaración menciona LIST como un vector (o un arreglo unidimensional) cuando mucho de 100 valores enteros, llamados LIST[1], LIST[2], ..., LIST[100]. Por tanto, el programa podrá manejar hasta 100 elementos contenidos en la lista.

La declaración de LIMIT como un valor entero único se utilizará para indicar cuántos de los 100 posibles elementos de la lista se están utilizando en realidad en un momento dado.

Un método para determinar cuándo se han ordenado todos menos uno de los valores originales se vale de la variable entera LAST (que se deberá declarar con LIMIT) como marcador de lugares. Los elementos de LIST[1] a LIST[LAST] estarán desordenados, y los elementos de LIST[LAST + 1] a LIST[LIMIT] estarán ordenados y serán mayores que todos los elementos

```

While "condición" do
  --
  --
  instrucciones por ejecutar en tanto "condición" sea verdadera
  --
  --
end while

for counter: = "valor inicial" to "valor final" do
  --
  --
  instrucciones por ejecutar con el contador puesto a los valores
  del intervalo
  --
  --
end for

if "condición" then
  --
  --
  instrucciones por ejecutar en caso de que "condición" sea
  verdadera
  --
  --
else
  --
  --
  instrucciones por ejecutar en caso de que "condición" sea falsa
  --
  --
end if

```

Fig. 58-1. Estructuras básicas de control en pseudocódigo.

no ordenados. Inicialmente LAST será LIMIT, para indicar que todos los elementos están desordenados. En la figura 58-3 se muestra el pseudocódigo de un método de burbuja inicial.

La notación "[:=" significa "hacer la variable de la izquierda igual al contenido de la variable de la derecha o al valor de la constante del lado derecho". La instrucción "while" es una orden repetitiva que ejecutará todas las instrucciones situadas entre la instrucción "while" inicial y "end while" final, en tanto la condición sea "verdadera". La condición "verdadera" corresponde a $LAST > 1$. La condición se volverá por último "falsa", ya que LAST se está reduciendo en 1 cada vez que se ejecuta el contenido de la instrucción while. La disminución de LAST en 1 se expresa de manera simbólica como $LAST := LAST - 1$ (fig. 58-4).

La porción de la instrucción while que hace ascender el elemento más grande hasta el final implica la comparación de elementos adyacentes y su intercambio cuando estén fuera de orden. Por tanto la instrucción while de la figura 58-3 se afina hasta obtener la forma de la figura 58-4.

El recorrido a través del arreglo LIST de LIST[1] a LIST[LAST - 1] se realiza por medio de una instrucción "for", que hace posible que una variable entera, I, tome valores sucesivos desde 1 hasta LAST - 1, inclusive. Esta variable se emplea como subíndice de LIST, y se incrementa en 1 en el terminador "end for". El intercambio se realiza en tres pasos mediante el uso de

una variable entera temporal, TEMP. TEMP e I deberán declararse con LIMIT y LAST. El refinamiento final se ilustra en la figura 58-5.

El contenido de la porción "then" de la instrucción "if" se ejecuta entre "if" y el terminador "end if" solamente cuando la condición es "verdadera". De lo contrario no se ejecuta la parte "then" de la instrucción. Los elementos de la lista se pueden imprimir mediante el uso de una instrucción "for", como se muestra en la figura 58-6.

Por último, todas las piezas se conjuntan para obtener el programa en pseudocódigo completo, como se ilustra en la figura 58-7. Obsérvese el principio de la pareja inicio-final para delimitar todo el programa, y el pareamiento de las estructuras de control "if", "while", y "for" con sus terminadores correspondientes.

58.2.2 Ejecución de un programa

Después de haber diseñado y escrito un programa en un lenguaje de alto nivel, el siguiente paso es ejecutarlo (corregirlo). Esto se realiza en cuatro pasos:

1. Compilación del programa en "código de máquina".
2. Enlaces (vinculación) de las piezas o partes del programa.
3. Carga del "código objeto" resultante en la memoria de la computadora.
4. Ejecución del programa.

Compilación. El proceso de compilación lo realiza un programa llamado compilador, el cual traduce a "código objeto" el programa "fuente" escrito en lenguaje de alto nivel. La meta básica consiste en traducir todas las partes simbólicas del programa fuente (nombres de variables, rótulos de instrucciones, operadores como + o -, etc.) en números de la máquina (direcciones para variables y rótulos, "códigos de operación" de instrucciones de la máquina para operadores, etc.).

```

Var
  LIST : array [1..100] of integer;
  LIMIT : integer;

```

Fig. 58-2. Declaraciones.

```

LAST := LIMIT;

while LAST > 1 do
  Desplazar el mayor de los elementos de LIST[1]
  al LIST[LAST] hasta el final de la lista (es decir,
  LIST[LIST]). Reducir LAST en 1.
end while

```

Fig. 58-3. Ordenador de burbuja inicial.

```
LAST := LIMIT;
```

```
while LAST > 1 do
```

```
  for "cada elemento entre LIST[1] y LIST[LAST - 1]" do
    comparar el elemento con el siguiente y, si no están
    en orden, intercambiarlos
```

```
  end for
```

```
  LAST := LAST - 1;
```

```
end while
```

Fig. 58-4. Refinamiento del método de ordenador de burbuja.

Al traducir un programa fuente, un compilador lo leerá por completo una o más veces. En cada "paso" el compilador realizará algunas de las tareas siguientes³:

- Análisis lexicológico.
- Análisis sintáctico.
- Manejo de tablas de símbolos.
- Generación de código.
- Optimización.

Algunos lenguajes, más notablemente el Pascal, se diseñaron para un compilador de "un paso", lo que significa que un compilador de Pascal leerá un programa Pascal sólo una vez y realizará todas las tareas mencionadas al mismo tiempo. Otros lenguajes, como el FORTRAN, suelen tener compiladores de "dos pasos". En el primer paso el objetivo consiste en realizar el análisis lexicológico y construir la tabla de símbolos. En el segundo paso se genera el código.

El análisis lexicológico divide el programa en "señales" o "prendas" indivisibles. Por ejemplo, en la instrucción de lenguaje Pascal:

```
PRECIO:= COSTO - DESCUENTO +
      + IMPUESTO
```

PRECIO se consideraría una señal única, := sería otra, COSTO otra, etc., hasta siete interpretaciones.

Después de realizar el análisis lexicológico, el compilador realiza un "análisis sintáctico" de las señales, lo cual asegura que éstas se dispongan en secuencias "legales" para el lenguaje que se está compilando. Si una instrucción no es sintácticamente correcta, el compilador imprime un mensaje de error a fin de indicarlo al programador.

El analizador sintáctico también agrupa las señales de instrucciones complejas en "subexpresiones" que reflejan el orden en el cual se realizarán las operaciones. En el caso de la instrucción de Pascal dada, por ejemplo, un compilador de Pascal estándar generaría un código que restara DESCUENTO de COSTO antes de sumar esa diferencia a IMPUESTO. Por otra parte, sería posible construir un analizador sintáctico que hiciera primero la suma y la restara de COSTO, lo que produciría resultados diferentes.

Este orden de "precedencia" de operaciones se define en forma inequívoca en cada lenguaje. Esto quiere decir que una instrucción legal dada de un lenguaje específico se puede analizar gramaticalmente en una sola forma, como lo determinan las "reglas sintácticas" de ese lenguaje. El compilador reconoce y traduce los programas que se apeguen a estas reglas.

La siguiente tarea que deberá realizar el compilador es la generación del código objeto que, cuando se ejecute, cumpla los objetivos del programa. El código objeto está en el mismo formato que el que produce un ensamblador y se puede considerar una serie de números que se cargarán en la memoria de la máquina.

Una instrucción individual de un lenguaje de alto nivel comúnmente se complica en una serie de varias instrucciones de la máquina. Esto se debe a que la mayor parte de las computadoras tienen instrucciones que solamente realizan tareas muy simples, mientras que los lenguajes de alto nivel suelen constituir herramientas conceptuales poderosas para solucionar problemas. Es labor del compilador traducir estas complejas herramientas a instrucciones simples; es decir, desglosar la operación compleja en diversos pasos.

El código generado por un compilador para un programa escrito en un lenguaje de alto nivel suele ser menos eficiente que el código de un programa en lenguaje ensamblador escrito para realizar la misma tarea. El código que generaban los primeros compiladores era notoriamente ineficiente, pero los más modernos se acercan mucho más al código "óptimo". La afinación del código para eliminar instrucciones redundantes se denomina optimización y puede realizarse por el programa en un paso extra del compilador.

```
LAST := LIMIT;
```

```
while LAST > 1 do
```

```
  for I := 1 to LAST - 1 do
    if LIST[I] > LIST[I + 1] then
      TEMP := LIST[I];
      LIST[I] := LIST[I + 1];
      LIST[I + 1] := TEMP;
```

```
    end if
```

```
  end for
```

```
  LAST := LAST - 1;
```

```
end while
```

Fig. 58-5. Refinamiento final del ordenador de burbuja.

```
for I := 1 to LIMIT do
```

```
  write (LIST[I]);
```

```
end for
```

Fig. 58-6. Instrucción "for".

Enlace (vinculación) y carga. Una vez que un programa fuente se ha compilado en código objeto, suele ser necesario "enlazarlo" ("vincularlo") y "cargarlo" en la memoria. Estas dos operaciones, enlace (o vinculación) y carga, a menudo son realizadas por el mismo programa, un "cargador de enlace", pero aquí se analizan por separado.

El enlazador une los módulos separados que pueda haber generado el compilador. En lenguajes como el FORTRAN, por ejemplo, cada subrutina o función termina como un "módulo objeto" separado. El problema principal que se observa al enlazar estos módulos objeto en un solo módulo de carga implica la modificación de las direcciones de las variables locales de cada módulo objeto para reflejar dónde figura ese módulo en relación con los otros módulos objeto.

La mayor parte de los compiladores facilitan esta modificación creando direcciones "relativas a cero" para rótulos de variables e instrucciones cuyas direcciones reales dependan de dónde se cargue el módulo en el cual figuran. Después, el enlazador modifica cada una de las direcciones relativas a cero de un módulo objeto agregándoles la suma de las longitudes de los módulos objeto que ya se han cargado.

Carga de un programa. El cargador simplemente copia en la memoria principal de la máquina el módulo de carga enlazado, de modo que el programa pueda por último ejecutarse. Su función principal consiste en indicar al sistema operativo la dirección donde comienza el módulo de carga y la longitud del módulo.

Ejecución de un programa. Después de que un programa se ha compilado, enlazado y cargado por fin está listo para ser "ejecutado" o "corrido". Lo único que necesita conocer el sistema operativo es la "dirección inicial" del programa. Ésta es la dirección física en la memoria de la primera instrucción ejecutable del procedimiento principal del programa. El programa "termina" cuando la ejecución de una instrucción genera una "trampa", y en ese momento el control de la máquina se retransfiere al sistema operativo.

58.3 SOFTWARE DE SISTEMAS

El concepto de software de sistemas comprende todos los programas de apoyo que suelen comprarse al distribuidor de computadoras en el momento en que se adquiere la computadora misma (este software puede estar incluido en un paquete con el hardware de la computadora, o tener que comprarse por separado). Una computadora sin software de sistemas es una herramienta en extremo deficiente para el programador de aplicaciones típico; las componentes que siguen suelen considerarse necesarias para el uso del hardware en un sistema generalmente útil:

- Ensamblador.
- Compiladores e intérpretes.
- Enlazador.
- Depurador.

Var

```
LIST : array [1..100] of integer;
LIMIT, LAST, I, TEMP : integer;

begin
  LIMIT := 0;

  while "no es el final del archivo" do
    LIMIT := LIMIT + 1;
    if LIMIT > 100 then
      write ("hay demasiados elementos en la lista.");
      halt;
    end if
    read (LIST[LIMIT]);
  end while

  if LIMIT = 0 then
    halt;
  end if

  LAST := LIMIT;
  while LAST > 1 do
    for I := 1 to LAST - 1 do
      if LIST[I] > LIST[I + 1] then
        TEMP := LIST[I];
        LIST[I] := LIST[I + 1];
        LIST[I + 1] := TEMP;
      end if
    end for
  end while

  for I := 1 to LIMIT do
    write (LIST[I]);
  end for
end.
```

Fig. 58-7. Versión final del programa ordenador de burbuja en pseudocódigo.

- Sistemas de manejo y editores de archivos.
- Unidades de entrada y salida.
- Sistema operativo.
- Sistema del lenguaje de control.

Ensambladores, compiladores e intérpretes. Traducen programas de computadora simbólicos al lenguaje binario del hardware de la computadora misma. Estos programas son quizá las herramientas que más trabajo ahorran en todo el repertorio de la computadora. Además de proporcionar una notación para utilizarse al nivel del ser humano o de la aplicación, muy a menudo están equipados con bibliotecas de programas para el tiempo de la ejecución (corrida) a fin de realizar procedimientos comunes, como evaluación de funciones matemáticas, ordenamiento, y formateo de la entrada y la salida.

Enlazador. Es un programa de computadora que combina en una sola unidad para su ejecución las componentes del programa, como un programa principal, subrutinas y componentes de la biblioteca del sistema. Esto permite un uso automatizado de las bibliotecas

proporcionadas por el vendedor, las instalaciones locales de apoyo a la computadora o los programadores individuales del equipo encargado de un proyecto. Estas bibliotecas hacen posible la fácil construcción y uso de las componentes de software activas y la madurez de la construcción del software en una disciplina parecida a la ingeniería.

Depurador (debugger). Es un sistema especializado de prueba de programas que permite a un programador verificar un programa en desarrollo, probarlo en un subconjunto seleccionado de sus pasos, interrogar y modificar sus variables, medir su rendimiento, y detectar y eliminar los errores que tenga.

Unidades de entrada y salida. Se utilizan para controlar los periféricos de una computadora: lectoras y perforadoras de tarjetas, unidades de cinta magnética y de papel, impresoras, unidades de disco duro y flexible (o disquete), terminales de usuarios y cualquier equipo que vincule la computadora con el mundo exterior. Estos complicados procesos se desarrollan una vez y son utilizados en todas las aplicaciones.

Sistemas de manejo y editores de archivos. Constituyen un medio adecuado de almacenamiento, recuperación y modificación de programas y datos. El almacenamiento de discos en línea y el de cintas magnéticas y discos fuera de línea de la computadora conforman un sistema de archivo óptimo y mucho más flexible que las gavetas de archivo de sistemas alimentados de papel. Los sistemas de manejo de archivo se extienden, en potencia, hasta convertirse en elaborados "sistemas de bases de datos" de referencia cruzada (que se describen más adelante), y los editores de archivos se extienden hasta transformarse en "procesadores de palabras", que hacen posible que las máquinas de composición de documentos produzcan trabajos terminados actualizados tan fácilmente como podrían formar dibujos en borrador o dibujos marcados con tipos mediante el uso de sistemas alimentados de papel.

Sistemas operativos. Ayudan a los usuarios de computadoras administrando por ellos los recursos del sistema. Entre estos recursos suelen contarse los dispositivos de entrada y salida [(I/O), de manera que muchos usuarios puedan imprimir sin tener que dejar su salida codificada], la memoria principal y la auxiliar, y la unidad de procesamiento central (CPU) de la computadora misma. El sistema operativo comparte estos recursos a fin de que una sola computadora opere como si fueran varias y atienda a diversos usuarios de manera concurrente, por lo que las docenas de usuarios de terminales de tiempo compartido tienen la impresión de ser los únicos usuarios del sistema.

Sistema del lenguaje de control. Es el procesador que interpreta las solicitudes de los usuarios y las transmite al sistema operativo. Los usuarios especifican, en el lenguaje de control, lo que pretenden hacer en la computadora: compilar un programa, ejecutar un programa con ciertos datos, imprimir un archivo, etc.

58.4 LENGUAJES DE PROGRAMACIÓN

58.4.1 Panorama general

Un lenguaje de programación permite al usuario representar un algoritmo en una forma que sea significativa para la computadora. Los problemas y, por tanto, los algoritmos para resolver los problemas varían en forma considerable. A partir de la necesidad de expresar algoritmos muy diferentes se ha originado una gran variedad de lenguajes de programación que varían considerablemente en tipo y capacidad. El profesional bien informado define el problema, determina un algoritmo para resolverlo y después evalúa los méritos de los diversos lenguajes de programación de que dispone. Entonces elige el que mejor se adapte al problema. Para tomar una decisión, deben considerarse diversos aspectos de un lenguaje.

Una división importante de los lenguajes de programación es: lenguajes de bajo nivel y de alto nivel. Los lenguajes de bajo nivel tienden a ser muy parecidos al código de la máquina. Esto los hace difíciles de entender, y requieren mucho trabajo de programación para realizar tareas mínimas. El lenguaje de bajo nivel más común es el ensamblador, que es específico para cada máquina. Los programas escritos en lenguaje ensamblador por lo general no se pueden transportar a otras máquinas.

Los lenguajes de alto nivel tienden a ser "amables con el usuario" y razonablemente fáciles de leer. Una sola instrucción de un lenguaje de alto nivel puede realizar el trabajo de muchas instrucciones de lenguaje ensamblador. Algunos lenguajes de alto nivel bien conocidos son: FORTRAN, COBOL, BASIC, PL/I, SNOBOL y Pascal. Incluso las órdenes que acepta el sistema operativo de una computadora podrían considerarse lenguaje de alto nivel. A este respecto, el sistema operativo UNIX* tiene un conjunto de instrucciones en extremo poderosas.

Los lenguajes de programación se pueden clasificar, en términos generales, como traducidos o interpretados. Comúnmente, un programa escrito en un lenguaje traducido se somete a un compilador o ensamblador, y se produce otra versión del programa. La nueva versión, llamada archivo objeto, se escribe en el código de la máquina y no es inteligible en absoluto salvo para la computadora. El archivo objeto se carga y ejecuta cuando se necesita el programa. FORTRAN, COBOL, C, PL/I y Ada son algunos de los lenguajes traducidos más notables. En general, los lenguajes compilados requieren declaraciones de variables y realizan una verificación de tipo estático. Sin embargo, muchos lenguajes compilados permiten algún tipo de asignación dinámica del almacenamiento.

Un programa escrito en un lenguaje interpretado no se envía a través de un compilador o ensamblador. En lugar de ello, se carga el intérprete en la memoria principal junto con el programa. Entonces se interpreta el programa línea por línea conforme se ejecuta, y el intérprete envía a la computadora el código de máqui-

* UNIX es una marca registrada de Bell Laboratories.

na adecuado. Típicamente, el lenguaje interpretado transporta descriptores del tiempo de la corrida, pero no requiere la declaración de variables. La mayor parte de los lenguajes interpretados realizan verificación de tipo dinámico, y por lo general todo el almacenamiento se asigna en forma dinámica. Algunos de los lenguajes interpretados mejor conocidos son BASIC, APL, LISP y SNOBOL.

La eficiencia de un lenguaje de programación se expresa en términos de velocidad en el momento de la ejecución, la cantidad de espacio que se necesita para operar el programa y la cantidad de trabajo de programación que se requiere. Los lenguajes traducidos son muy eficientes en el momento de la ejecución. Sin embargo, el precio de esta eficiencia es una pérdida de flexibilidad. Los lenguajes interpretados están diseñados para ofrecer la máxima flexibilidad del programa, pero por lo general se necesitan más tiempo y espacio para la ejecución.

Pocos lenguajes son compilados o interpretados "puros"; más bien, casi todos son un híbrido de los dos. Aunque el FORTRAN 77 se considera un lenguaje compilado, con frecuencia sus instrucciones de formato son implementadas por intérpretes I/O. Por otro lado, en la mayor parte de las implementaciones de SNOBOL, el programa escrito en este lenguaje se traduce a un código más "amable con la máquina" y después se interpreta el nuevo código. El Pascal es el híbrido real entre el lenguaje traducido y el interpretado. Algunas versiones del Pascal generan e interpretan el "código P" (un sistema de programación para una computadora Pascal mítica), y otras traducen después el código P al lenguaje de la máquina en ejecución.

Otras propiedades de un lenguaje que deben considerarse cuando se juzgue la usabilidad del lenguaje son:

- Las estructuras de datos que ofrece.
- Su portabilidad.
- La cantidad de documentación disponible.
- La brevedad de la definición del lenguaje.
- Su idoneidad para resolver un problema.
- La disponibilidad de un compilador o intérprete.

58.4.2 Lenguajes específicos

Aquí se describen diversos lenguajes de programación por su transcendencia histórica, puntos fuertes y referencias. Se presenta un programa como ejemplo, el de ordenamiento de burbuja, como punto de comparación. También se presenta un segundo programa como ejemplo para mostrar las facultades específicas de cada lenguaje, en especial de los lenguajes más especializados.

Algol. Algol es el producto del trabajo de un grupo internacional de científicos especializados en computación. Inicialmente llamado International Algebraic Language (lenguaje algebraico internacional), se hizo público en los informes Algol 58 y Algol 60. Entre las características por las que el Algol se hizo famoso se encuentra el hecho de que en un principio se definió en

forma de metalenguaje, que después se conoció como BNF, y fue el vehículo para hacer que este metalenguaje fuese ampliamente conocido y aceptado^{4,5}. El lenguaje era simple (sólo tenía seis tipos de instrucciones y unas cuantas restricciones). Por ejemplo, los identificadores podían ser arbitrariamente largos, y un arreglo podía tener un número cualquiera de dimensiones. No obstante, el lenguaje era poderoso, en el sentido de que incluía estructura de bloques, funciones y procedimientos recurrentes, paso por valor y paso por nombre. Originalmente el lenguaje no tenía facilidades de entrada y salida, aunque éstas las proporcionaron después los procedimientos. Se esperaba que todas las funciones y procedimientos definidos por el usuario se definieran internamente en el programa solicitante. Con esta característica, el lenguaje Algol se podía llamar altamente tipificado. Fue diseñado para realizar cálculos científicos, y fue el lenguaje de publicación de algoritmos en la comunidad ACM (Association for Computing Machinery). A pesar de ser popular en Europa, Algol nunca suplantó a FORTRAN en Estados Unidos. No obstante, Algol, su metalenguaje y los esfuerzos de investigación realizados en la implementación de Algol han tenido un profundo efecto en la computación y en lenguajes posteriores, como el PL/I, Pascal y Ada, inigualado quizá por ningún otro lenguaje. En la figura 58-8 se presenta la versión en Algol del programa ordenador de burbuja.

COBOL. Durante los últimos años de la década de 1950, el Department of Defense y otras agencias del gobierno de Estados Unidos estaban concediendo contratos. Los lenguajes de programación que se utilizaban para estos contratos variaban desde el alto al bajo nivel. Esto volvió extremadamente costoso el trabajo de seguir los contratos y mantener los programas, ya que se tenía que capacitar a auditores y programadores para trabajar con una amplia variedad de lenguajes de programación. En consecuencia, se adoptó la idea de un lenguaje común para realizar todos los contratos del gobierno. En 1959 se creó el Committee On Data Systems Languages, CODASYL, integrado por representantes de las agencias del gobierno de Estados Unidos, fabricantes de computadoras, usuarios de computadoras y universidades. Este comité decidió crear un nuevo lenguaje debido a que la mayor parte de los existentes en ese tiempo no se adaptaban bien a la tarea del procesamiento de datos. Las diferencias del hardware de los fabricantes de computadoras también contribuyeron a la dificultad de adoptar algún lenguaje de esa época. El nuevo lenguaje, COBOL (*common business-oriented language*, lenguaje común orientado a las empresas), tenía el propósito de que lo entendieran personas con escasos conocimientos de computación^{6,7}.

Las especificaciones iniciales se publicaron en el mes de abril de 1960, y se hicieron revisiones en 1963 y 1965. Para preservar las características comunes del COBOL entre los diferentes fabricantes, el American National Standards Institute (ANSI) publicó, en 1968, sugerencias para hacer del COBOL un lenguaje de programación estándar. Por tanto, se puso a dispo-

```

procedure demo begin
real array X[1 : 100];

procedure bubble (A, FIRST, LAST); real array A[FIRST : LAST];
begin integer I; real TEMP; boolean SWAPPED;
  LOOP : SWAPPED := false;

  for I := FIRST step 1 until LAST - 1 do

    if A[I] > A[I + 1] then begin
      TEMP := A[I];
      A[I] := A[I + 1];
      A[I + 1] := TEMP;
      SWAPPED := true; comment: record that a swap was needed;
    end;

  if SWAPPED then go to LOOP; comment: repeat loop;
end;

read(X); bubble (X, 1, 100); write (X)
end

```

Fig. 58-8. Programa ordenador de burbuja en Algol.

sición una norma para la construcción de compiladores de COBOL. Los compiladores de COBOL que se apegan a la norma se conocen como compiladores ANSI COBOL. En 1974, la ANSI revisó las especificaciones estándares de COBOL y actualmente están en revisión nuevas normas.

COBOL es el lenguaje de computadora más ampliamente utilizado en el mercado en la actualidad. Algunas estimaciones indican que en el 80% de las nuevas aplicaciones de las empresas se utiliza el lenguaje COBOL. Este lenguaje debe su vasto uso a la adopción que hicieron de él muchos contratistas bajo la insistencia del gobierno; su estandarización por parte de la ANSI; sus instrucciones parecidas a las del idioma inglés, que son más fáciles de leer que las de otros lenguajes; su gran cantidad de características de edición de datos; sus facilidades para el manejo de archivos; y por último, su naturaleza evolutiva a medida que cambian las necesidades de la comunidad empresarial. Las principales deficiencias de COBOL son su verbosidad, la carencia de capacidades matemáticas y la carencia de construcciones estructuradas, que sí se hallan en el Algol o PL/I. Sin embargo, con la disciplina apropiada, los programas escritos en COBOL puede ser razonablemente bien estructurados.

El programa de ordenamiento de burbuja implementado en COBOL se muestra en la figura 58-9. Todos los programas escritos en COBOL deberán tener las cuatro divisiones en el orden que se presenta. La división de identificación da el nombre del programa y otros datos de identificación. (En la figura se han omitido varios elementos opcionales con fines de brevedad y claridad.) La división del entorno contiene características basadas en el hardware de un fabricante de computadoras, en especial los nombres de los dispositivos y archivos físicos que se asociarán con los nombres lógicos del COBOL (la instrucción SELECT-ASSIGN).

La división de datos define la estructura de todos los archivos y el tipo y tamaño de todos los identificadores (variables) que se usan en el programa. La cláusula PIC (o imagen, de "picture") define el tamaño y tipo [el tipo por medio de los símbolos 9 (numéricos) y X (alfanumérico)] y la longitud por medio del número de estos símbolos. El símbolo "S" indica que se almacene el signo del número, mientras que "V" indica ubicación implícita del punto decimal cuando los datos no lo contienen. Un elemento que se utilice para realizar aritmética sólo puede contener S, V y 9. La estructura jerárquica de los datos se implementa con el número de nivel. Un registro deberá comenzar con el nivel 01. Un campo subordinado emplea un número mayor. Por tanto, DETAIL-LINE se compone de dos elementos: FILLER y DATA-VALUE-OUT. La palabra FILLER es reservada; es decir, es una palabra de la sintaxis del COBOL que tiene un significado especial. Se utiliza cuando un campo no necesita indicarse directamente por nombre. Este campo se indica en forma implícita cuando se hace referencia a DETAIL-LINE. DATA-VALUE-OUT es un campo editado, o sea, un campo que se utiliza para editar un número en una forma que se pueda imprimir. Este campo tiene un signo menos (—) si es negativo o un espacio en blanco si es positivo, seguido de cuatro cifras (dígitos) sin ceros, un punto decimal y dos espacios decimales. El nivel 88 tiene un uso especial e indica un "nombre de condición". El nombre de condición tiene un valor verdadero/falso, dependiendo de si el identificador precedente contiene el valor que se da en la entrada del nivel 88.

La división del procedimiento contiene el algoritmo para utilizar los datos. Las instrucciones se agrupan en párrafos con nombre asignado que pueden referirse por medio de las instrucciones PERFORM. La instrucción PERFORM hace que las instrucciones del párrafo que se especifica se ejecuten una vez, o en forma re-

IDENTIFICATION DIVISION.
PROGRAM-ID. BUBBLE-SORT.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

 SELECT UNSORTED-FILE ASSIGN TO "DATAFILE".
 SELECT SORTED-REPORT ASSIGN TO PRINTER.

DATA DIVISION.

FILE SECTION.

FD UNSORTED-FILE.

01 DATA-VALUE-RECORD.

 05 DATA-VALUE-IN PIC S9999V99.

FD SORTED-REPORT.

01 REPORT-LINE PIC X(133).

WORKING-STORAGE SECTION.

01 FLAGS.

 05 MORE-DATA-REMAINS-FLAG PIC X VALUE "Y".

 88 NO-MORE-DATA-REMAINS VALUE "N".

 05 SWITCH-FLAG PIC X VALUE "Y".

 88 NO-VALUES-SWITCHED VALUE "N".

01 NUMBER-TABLE.

 05 NO-VALUES PIC S999 USAGE IS COMPUTATIONAL.

 05 DATA-VALUE PIC S9999V99 OCCURS 100 TIMES

 USAGE IS COMPUTATIONAL-3.

 05 DATA-SUB PIC S999 USAGE IS COMPUTATIONAL.

 05 DATA-TEMP PIC S9999V99 USAGE IS COMPUTATIONAL.

01 HEADINGS.

 05 FILLER PIC X(14) VALUE "SORTED VALUES".

01 DETAIL-LINE.

 05 FILLER PIC XXX.

 05 DATA-VALUE-OUT PIC -ZZZ9.99.

PROCEDURE DIVISION.

MAIN-LINE-ROUTINE.

 PERFORM READ-ROUTINE.

 PERFORM SORT-ROUTINE UNTIL NO-VALUES-SWITCHED.

 PERFORM WRITE-ROUTINE.

 STOP RUN.

READ-ROUTINE.

 OPEN INPUT UNSORTED-FILE.

 MOVE ZERO TO DATA-SUB.

 READ UNSORTED-FILE AT END MOVE "N" TO MORE-DATA-REMAINS-FLAG.

 PERFORM READ-A-VALUE UNTIL NO-MORE-DATA-REMAINS.

 MOVE DATA-SUB TO NO-VALUES.

 CLOSE UNSORTED-FILE.

READ-A-VALUE.

 ADD 1 TO DATA-SUB.

 MOVE DATA-VALUE-IN TO DATA-VALUE (DATA-SUB).

 READ UNSORTED-FILE AT END MOVE "N" TO MORE-DATA-REMAINS-FLAG.

SORT-ROUTINE.

 MOVE "N" TO SWITCH-FLAG.

 PERFORM SORT-PASS VARYING DATA-SUB FROM 1 BY 1

 UNTIL DATA-SUB > NO-VALUES - 1.

Fig. 58-9. Programa ordenador de burbuja en COBOL.

```

SORT-PASS.
  IF DATA-VALUE (DATA-SUB) > DATA-VALUE (DATA-SUB + 1)
    MOVE DATA-VALUE (DATA-SUB) TO DATA-TEMP
    MOVE DATA-VALUE (DATA-SUB + 1) TO DATA-VALUE (DATA-SUB)
    MOVE DATA-TEMP TO DATA-VALUE (DATA-SUB + 1)
    MOVE "Y" TO SWITCH-FLAG.

WRITE-ROUTINE.
  OPEN OUTPUT SORTED-REPORT.
  WRITE REPORT-LINE FROM HEADINGS AFTER ADVANCING PAGE.
  PERFORM WRITE-A-VALUE VARYING DATA-SUB FROM 1 BY 1
    UNTIL DATA-SUB > NO-VALUES.
  CLOSE SORTED-REPORT.

WRITE-A-LINE.
  MOVE SPACES TO DETAIL-LINE.
  MOVE DATA-VALUE (DATA-SUB) TO DATA-VALUE-OUT.
  WRITE REPORT-LINE FROM DETAIL-LINE AFTER ADVANCING 1 LINES.

```

Fig. 58-9 (cont.). Programa ordenador de burbuja en COBOL.

petida, hasta que una condición sea verdadera. La frase AT END de la instrucción READ se ejecuta solamente cuando se encuentra una condición de final de archivo.

Un programa de aplicación COBOL típico no tendría un ordenamiento codificado con ciclos, como se muestra en la figura 58-9, sino que en cambio se utilizaría la instrucción SORT^{6,7}. La instrucción SORT utiliza un archivo sin ordenar como entrada a una rutina de ordenamiento. El archivo de salida contendría los registros ordenados, que se leerían y procesarían. Una entrada de descripción del ordenamiento (SD, de *sort description*) de la sección de archivos especifica la longitud del registro por ordenar y si el archivo se ordenará en orden ascendente o descendente.

FORTRAN IV y FORTRAN 77. El lenguaje FORTRAN (*formula translator*, traductor de fórmulas)⁸, desarrollado por IBM a mediados de la década de 1950 principalmente para aplicaciones científicas y de ingeniería, fue el primer lenguaje de alto nivel ampliamente usado. Sus facultades más importantes son una ejecución muy eficiente y la facilidad con la cual se puede enlazar los módulos. Esta provisión de compilación aparte hace posible la generación de una biblioteca de rutinas que se puede utilizar en la producción de nuevo software.

La estructura básica de un programa FORTRAN es simplemente el módulo principal seguido de subprogramas internos. Los subprogramas externos no necesitan recompilarse, sino que se pueden enlazar, siempre que sean referidos por una instrucción EXTERNAL o por una instrucción CALL explícita.

El conjunto de caracteres está limitado a letras mayúsculas, cifras y los nueve caracteres especiales:

, . () ' + - / =

En el código fuente, las columnas 1 a 5 forman un campo de rótulo numérico, la sexta posición indica una línea de continuación y las columnas 7 a 72 son para las instrucciones en sí. Algunos compiladores relajan este

convencionalismo (orientado a las tarjetas) y permiten el uso de un formato más libre (que se utiliza con equipo moderno).

Dado que en el diseño se dio mayor importancia a la ejecución eficiente, existen estrictas limitaciones en cuanto al número de tipos de datos y estructuras de control. Hay cinco tipos de datos: enteros, reales, complejos, de doble precisión y lógicos (booleanos). Las estructuras de datos están limitadas a variables simples y arreglos de hasta tres dimensiones. Las variables numéricas se representan de manera implícita por medio de su primera letra, de la I a la N para variables enteras y las otras letras para variables reales; o bien de manera explícita por medio de instrucciones de declaración como REAL I, J o INTEGER A. Las variables lógicas se deben representar en forma explícita con una instrucción de declaración LOGICAL. Los arreglos se definen en una instrucción de declaración o en una instrucción DIMENSION, indicando el número de entradas de cada dimensión. Por ejemplo, DIMENSION A(3,4) y REAL I(10) definen arreglos reales de dos y una dimensión; A es un arreglo cuyos subíndices van de 1 a 3 y de 1 a 4, e I es un arreglo cuyo (único) subíndice va de 1 a 10.

Existen tres estructuras de control básicas: GOTO, una ramificación incondicional; IF, una ramificación condicional; y DO, un constructo de repetición. Son posibles los subprogramas internos y externos. Éstos son módulos separados y tienen como encabezado una instrucción FUNCTION o SUBROUTINE. El acceso es por medio de una instrucción CALL (llamada) para subrutinas y mediante el uso del nombre para funciones. En una instrucción EXTERNAL deberán citarse los subprogramas externos. Los argumentos se pueden transmitir por medio de una lista de argumentos o indicando el alcance global de la variable por medio de una instrucción COMMON en el módulo solicitante y en el subprograma. Para mejorar la eficiencia, se prohíbe el uso de la recursión (es decir, una subrutina que se llama a sí misma).

La entrada y salida son manejadas por las instrucciones READ y WRITE, respectivamente. Con cual-

quiera de estas instrucciones, el programador especifica un dispositivo por medio de un número (impresora, lectora de tarjetas, cinta o archivo de disco), un número de formato y una lista de variables del programa que se leerán o escribirán. La instrucción **FORMAT** especifica la forma en que el programador desea que se realice la conversión entre los caracteres binarios internos de la computadora y los caracteres externos legibles.

El programa ordenador de burbuja escrito en **FORTRAN IV**⁸ se presenta en la figura 58-10.

Las limitaciones del **FORTRAN IV** condujeron al desarrollo del **FORTRAN 77**⁹⁻¹¹, que ahora es la versión confirmada del lenguaje. Entre las nuevas características importantes se cuentan la adición del constructo **IF-THEN-ELSE**, la posibilidad de dimensionamiento dinámico de arreglos y la adición del tipo de datos **CHARACTER**. En la figura 58-11 se presenta el programa ordenador de burbuja escrito en **FORTRAN 77**.

PL/I. El desarrollo de **PL/I** comenzó en 1963, cuando un grupo de usuarios de **IBM** formaron un subcomité llamado **Advanced Language Development Committee**. El objetivo original de este comité era diseñar un lenguaje que sucediera al **FORTRAN**. Su trabajo fue apoyado totalmente por **IBM**, que buscaba un nuevo lenguaje para hacer un mejor uso de sus nuevas computadoras todavía no anunciadas.

El comité invitó a muchos expertos de todo el mundo a sugerir características que pudieran formar parte de este nuevo lenguaje. Además, los miembros del comité examinaron lenguajes existentes con el fin de determinar cuáles de sus características podrían incorporarse al nuevo lenguaje. Por tanto, lo que comenzó como una extensión del **FORTRAN** rápidamente se convirtió en un lenguaje completamente diferente.

Este nuevo lenguaje se llamó inicialmente **NPL**, de *new programming language* (nuevo lenguaje de programación). Sin embargo, se desechó este nombre debido a un conflicto con el **National Physics Laboratory** de Inglaterra. Finalmente, se adoptó el nombre **PL/I**, de *programming language I* (lenguaje de programación I). La implementación de este lenguaje se inició en **IBM** en Inglaterra en 1964. El primer manual se publicó en 1965 y el primer compilador se lanzó al mercado en 1966.

Como podría esperarse de un lenguaje diseñado por un comité, el **PL/I** es una combinación de muy diversas construcciones y características. Hasta entonces, los lenguajes de programación se habían diseñado para funcionar más efectivamente en áreas de aplicación específicas (p. ej., el **COBOL** para medios comerciales, **FORTRAN** para ciencias e ingeniería). El **PL/I** se diseñó con muchas características de éstos y otros lenguajes. Se pensó que **PL/I** era lo suficientemente general para ser utilizado en la solución de problemas de un medio cualquiera¹².

Quizá la característica más prominente de **PL/I** es su uso liberal de características de omisión (default). Virtualmente todos y cada uno de los tipos de construcción de programación que pueden presentarse al

```

READ (5, 500) A
500  FORMAT (I10)
      CALL BUBBLE (FIRST, LAST)
      WRITE (6, 600) A
600  FORMAT (1H, 100(I5, 2X))
      STOP
      END

SUBROUTINE BUBBLE (FIRST, LAST)
  INTEGER FIRST, I, LAST, TEMP, A(10)
  COMMON A
  LOGICAL SWAPPD

10    SWAPPD = .FALSE.
      DO 100 I = FIRST, LAST - 1
        IF (.NOT.(A(I).GT.A(I + 1))) GOTO 100
        TEMP = A(I)
        A(I) = A(I + 1)
        A(I + 1) = TEMP
        SWAPPD = .TRUE.

100   CONTINUE
      IF (SWAPPD) GOTO 10
      RETURN
      END

```

Fig. 58-10. Programa ordenador de burbuja en **FORTRAN IV**.

```

INTEGER A(10)
READ (5, 500) A
500  FORMAT (I10)
      CALL BUBBLE (1, 10, A, 10)
      WRITE (6, 600) A
600  FORMAT (1H, 100(I5, 2X))
      STOP
      END

SUBROUTINE BUBBLE (FIRST, LAST, A, SIZE)
  INTEGER FIRST, I, LAST, TEMP, A(SIZE)
  LOGICAL SWAPPD

10    SWAPPD = .FALSE.
      DO 100 I = FIRST, LAST - 1
        IF (A(I).GT.A(I + 1)) THEN
          TEMP = A(I)
          A(I) = A(I + 1)
          A(I + 1) = TEMP
          SWAPPD = .TRUE.
        END IF

100   CONTINUE
      IF (SWAPPD) GOTO 10
      RETURN
      END

```

Fig. 58-11. Programa ordenador de burbuja en **FORTRAN 77**.


```

SORTER: PROCEDURE OPTIONS (MAIN);
DECLARE I, B(1 : 10) FIXED (31);

BUBBLE: PROCEDURE (A, LO, HI);
DECLARE I, TEMP, A(*), LO, HI FIXED (31),
        SWAPPED BIT (1);

        SWAPPED = '0'B; /*INICIALIZAR SWAPPED A FALSO.*/
        DO WHILE (~ SWAPPED);
            DO I = LO TO HI - 1;
                IF A(I) > A(I + 1) THEN DO;
                    TEMP = A(I);
                    A(I) = A(I + 1);
                    A(I + 1) = TEMP;
                    SWAPPED = '1'B; /*REGISTRAR EL INTERCAMBIO.*/
                END;
            END;
        END;
END BUBBLE;

/*EL PROGRAMA PRINCIPAL COMIENZA AQUÍ.*/
DO I = 1 TO 10; /*ENTRADA*/
    READ(B(I));
END;
CALL BUBBLE (B, 1, 10); /*ORDENAR B*/

DO I = 1 TO 10; /*SALIDA*/
    WRITE(B(I));
END;

END SORTER;

```

Fig. 58-12. Programa ordenador de burbuja en PL/I.

compilador tienen un valor de omisión correspondiente. Los creadores razonaron que, como la tecnología de compiladores había avanzado en forma importante desde los primeros días de los lenguajes FORTRAN y COBOL, y las máquinas eran cada vez más poderosas, esta potencia se debía aprovechar para facilitar la vida a los programadores. Lo que aquéllos no apreciaron fue la forma en la cual las características de omisión pueden en realidad complicar el proceso de desarrollo de programas. Los programadores de PL/I tienen que aprender todas las posibilidades de omisión, a fin de no introducir errores en su código debido a una suposición hecha por el compilador.

PL/I fue uno de los primeros lenguajes de alto nivel en ofrecer libre acceso a recursos del sistema operativo en el momento de la ejecución. Ofrece medios a los programadores para manipular con facilidad la mayor parte de las condiciones en el momento de la ejecución. En casi todos los otros lenguajes de alto nivel, todos los errores y condiciones en el momento de la ejecución se manejan poniendo fin al programa del usuario. Si bien PL/I está diseñado para ser independiente de la máquina, tiene cierta inclinación a depender del sistema operativo. Los programas de usuarios codificados para utilizar las características de un sistema operativo específico no se ejecutarán en un sistema diferente.

Otras características interesantes de PL/I son: un equipo de manejo de macroinstrucciones para el mo-

mento de la compilación, tipos de datos en cadena (de caracteres y bits) y tipos de datos numéricos de precisión variable.

Cuando se propuso por vez primera el PL/I se pensó que era el lenguaje del futuro, y los primeros compiladores se esperaron con impaciencia. A medida que se hicieron evidentes las deficiencias de este potente lenguaje, los usuarios se volvieron renuentes a cambiar a PL/I. A diferencia del FORTRAN y COBOL, que son estandarizados y los apoya una gran cantidad de hardware de los fabricantes, el PL/I es apoyado por unos cuantos distribuidores además de IBM. Los usuarios no están impacientes por invertir grandes cantidades de capital en software que los ataría al hardware de un fabricante. El programa ordenador de burbuja escrito en PL/I se muestra en la figura 58-12.

BASIC. El lenguaje de computadora BASIC (*beginner's all-purpose symbolic instruction code*, código de instrucciones simbólico de uso general para principiantes) se originó en el Dartmouth College a mediados de la década de 1960 bajo la dirección de John G. Kemeny y Thomas E. Kurtz¹³. Su objetivo era diseñar un lenguaje que fuera sencillo de aprender y utilizar para programadores inexpertos a fin de resolver problemas de programación simples. El lenguaje original se diseñó con un número mínimo de estructuras de control y estructuras de datos. Los tres primeros caracteres de cada línea forman una palabra clave única, las reglas

de sintaxis de todos los tipos de instrucciones son lo más idénticas posibles, y los nombres de variables son cortos. Estos factores hacen del BASIC fácil de usar. Las versiones originales del BASIC eran mínimas en naturaleza y estaban diseñadas como primer lenguaje para estudiantes. Después de que hubiesen dominado el BASIC, utilizarían otros lenguajes de computadora, principalmente el FORTRAN, para producir programas más rápidos y eficientes. En la figura 58-13 se muestra el programa ordenador de burbuja en lenguaje BASIC.

Cada instrucción comienza con un número de línea en orden ascendente. El primer elemento que está después del número de líneas es la palabra clave, que especifica el tipo de orden. La primera línea tiene la palabra clave REM, que especifica que esta línea es un comentario y no contiene ninguna orden. La instrucción DIM especifica que el arreglo unidimensional, llamado A (que se conoce como lista en el BASIC), tiene 50 elementos. El BASIC también tiene arreglos bidimensionales denominados tablas. La instrucción READ toma el siguiente número disponible del área de datos y lo almacena en la variable N (igual a 8). El área de datos está definida por las instrucciones DATA, que especifican un conjunto de números ordenado por almacenar en el área de datos interna. Los números se almacenan en el orden en que se teclean; todos los números de la instrucción DATA con la numeración más baja figuran antes de cualquier número en la siguiente instrucción DATA de numeración inferior. Las instrucciones FOR y NEXT especifican que las instrucciones que están entre ellas se ejecutarán un número de veces especificado, y el contador de ciclos (J y K) se incrementará en uno cada vez que se recorra el ciclo. La instrucción IF prueba la condición especificada [¿es A(K) menor o igual que A(K + 1)?] y si es verdadera, el control se transfiere al número de línea 140 (salta la instrucción para intercambiar los números). La instrucción LET asigna el valor de una expresión aritmética a la variable mencionada a la izquierda del símbolo "=". Las instrucciones MAT se utilizan para efectuar operaciones matriciales, es decir, operaciones con una lista o tabla completa. La instrucción MAT READ hace que los siguientes N (igual a 8) valores en el área de datos se almacenen en los elementos del 1 al 8 de la lista A [A(1) es 34.5, A(2) es 23.6, ..., A(8) es 44.2]. La instrucción MAT PRINT imprime todos los elementos de A a los que se haya asignado un valor.

Pascal. El Pascal fue diseñado y desarrollado por Niklaus Wirth. Se encuentra en la familia de lenguajes del Algol e incluye un vasto conjunto de estructuras de datos y posibilidades de construcción de estructuras de datos. En 1968 se produjo una versión preliminar, y la versión estándar se definió en 1974 en el *Pascal User Manual and Report*¹⁴. El IEEE Computer Standards Committee también ha publicado una norma¹⁵.

El lenguaje Pascal se diseñó principalmente como lenguaje de enseñanza. Como tal, contiene características que aceptan sin dificultad técnicas de programación estructurada, tales como la modularización y

```

10  REM ORDENAR UNA LISTA DE NÚMEROS
    UTILIZANDO EL MÉTODO DE LA BURBUJA
20  DIM A(50)
30  REM LEER EL NÚMERO DE VALORES (HASTA 50)
    POR ORDENAR
40  REM Y LOS VALORES DE LA LISTA DE DATOS
    INTERNA
50  READ N
60  MAT READ A(N)
70  REM EJECUTAR EL ORDENAMIENTO DE BURBUJA
    EN LA LISTA A
80  LET S = 0
90  FOR I = 1 TO N - 1
100     IF A(I) <= A(I + 1) THEN 140
110     LET T = A(I)
120     LET A(I) = A(I + 1)
130     LET A(I + 1) = T
135     LET S = 1
140  NEXT I
150  IF S = 1 THEN 80
160  REM IMPRIMIR LA LISTA
170  MAT PRINT A
180  DATA 8
190  DATA 34.5, 23.6, 0, 5.7, -87.5, 90, -0.7, 44.2
200  END

```

Fig. 58-13. Programa ordenador de burbuja en BASIC.

el diseño descendente. Entre éstas se incluyen una tipificación considerable, limitaciones sobre el campo de acción de variables y subprogramas, estructuras de control y de datos versátiles, y subprogramas que se pueden llamar en forma recursiva. Estas características y el hecho de que el Pascal es conciso a pesar de ser fácil de aprender y utilizar, constituyen sus ventajas más importantes. El lenguaje Pascal ganó la aceptación inmediata en la comunidad académica, y el éxito del método de programación estructurada ha contribuido asimismo a su creciente popularidad en los lugares de trabajo.

La estructura básica de un programa escrito en Pascal consta de una sección de declaración seguida de un bloque de instrucciones de acción que conforman el algoritmo apropiado. La sección de declaración es donde se definen y teclean constantes, tipos de datos definidos por el usuario, variables y estructuras de datos, y donde se colocan los subprogramas definidos por el usuario. En Pascal existen dos tipos de subprogramas principales, las funciones y los procedimientos. Las funciones se utilizan como variables, pero sus valores son determinados por la ejecución del subprograma. Los procedimientos se invocan por medio de una instrucción aparte y tienen el efecto de sustituir esa instrucción por el contexto del procedimiento. La estructura de los subprogramas es paralela a la de los programas principales. Cada subprograma tiene una sección de declaración en la que el programador puede definir elementos locales a ese subprograma. La sección de declaración va seguida de un bloque de instrucciones ejecutables que especifican las acciones del subprograma.

La entrada y salida se manejan más a menudo en términos de "archivos de texto", donde el archivo externo (por lo general una lectora de tarjetas, impresora

```

program bubble (input, output);
const  FIRST = 1;
      LAST = 10;
var A : array[FIRST..LAST] of integer;
    I : integer;

procedure bubble;
var I, temp : integer;
    SWAPPED : boolean;
begin
  repeat
    SWAPPED := false; {se supone que no se hicieron
      intercambios}
    for I := FIRST to LAST - 1 do
      if A[I] > A[I + 1] then
        begin {intercambiar dos elementos fuera de
          orden}
          temp := A[I];
          A[I] := A[I + 1];
          A[I + 1] := temp;
          SWAPPED := true {registrar que se necesitó
            un intercambio}
        end
      until not SWAPPED {¿concluyó el ordenamiento?}
    end;

begin {el programa principal comienza aquí}
  for I := FIRST to LAST do {introducir los elementos que se
    ordenarán}
    read (A[I]);
  bubble; {llamar al procedimiento de ordenamiento}

  for I := FIRST to LAST do {imprimir el arreglo ordenado}
    writeln (A[I])
  end.

```

Fig. 58-14. Programa ordenador de burbuja en Pascal.

o terminal interactiva) es legible para el operador y las variables internas del programa son binarias de la computadora. Los procedimientos integrados del lenguaje Pascal que realizan las operaciones de I/O son *read* (lectura) y *write* (escritura), con variaciones *readln* y *writeln*, que se leen al final de una línea de entrada y escriben toda una línea de salida, incluso el regreso de carro, respectivamente.

El programa ordenador de burbuja escrito en Pascal se muestra en la figura 58-14.

Las instrucciones se separan con el signo de punto y coma. En una línea puede haber más de una instrucción, y una instrucción se puede extender varias líneas. Todas las variables deben declararse, y es necesario especificar sus tipos de datos. Un concepto importante es el del campo de acción. El campo de acción de un elemento de un programa es el módulo en el cual se define, y todos los módulos textualmente contenidos en él. No está a la "vista" de los elementos del programa que lo incluyen textualmente. Así, en este ejemplo, como el arreglo A se declara en el programa principal, su campo de acción incluye el procedimiento "burbuja", y las referencias a A no causan problemas, de

modo que no es necesario declararla ahí. Sucedería lo mismo con la variable I del programa principal, salvo para la declaración de I en el procedimiento. Esto crea otra variable, que es local a la "burbuja" y evita que se haga referencia al valor del programa principal. El resultado son dos variables distintas con el mismo nombre.

Dos de las características más importantes del Pascal que no se utilizaron en el ejemplo anterior son los registros y los apuntadores¹⁶.

Registro. La estructura del registro puede considerarse una extensión del concepto de un arreglo, donde las entradas pueden ser de distintos tipos. Las entradas se refieren por el nombre del registro, y el nombre de la entrada se separa por un punto y no por el número de posición. Considérese el registro que se define en la figura 58-15. Si a una variable *widget* se le asigna el tipo *partrecord*, entonces *widget.parno* y *widget.price* se refieren al número de parte (pieza) y al precio.

Apuntador. Es una variable que hace referencia (o apunta) a una localidad del almacenamiento; su valor es la dirección de esa localidad. Para crear un apuntador, debe declararse un tipo de datos definido por el usuario como un apuntador al elemento que ocupará el área de almacenamiento, y después declarar variables de ese tipo. Para asignar valores a la variable apuntadora, se asignan localidades de almacenamiento empleando el procedimiento estándar *new* (nuevo) con la variable como argumento. Por ejemplo, para el acceso a registros de partes (piezas) como se han definido, por medio de apuntadores y no por nombres, se usan las declaraciones que se ilustran en la figura 58-16. Después las instrucciones *new (widget)* y *new (gizmo)* crean dos localidades de almacenamiento que contendrán la información sobre aquellas partes y colocarán las direcciones en *widget* y *gizmo*. Las construcciones *widget^.partno* y *gizmo^.price* se refieren al número de parte de *widget* y al precio de *gizmo*.

C. Éste es un lenguaje de programación de uso general creado originalmente por Dennis Ritchie para la PDP-11 de DEC, que corre (ejecuta) el sistema operativo UNIX*. El C no está sujeto a ningún sistema particular. Sin embargo, existen compiladores para otras máquinas, grandes y pequeñas. El lenguaje tiene sus orígenes en el lenguaje BCPL, por medio de un lenguaje intermedio llamado B (que se produjo en 1970 para el primer sistema UNIX, que corrió en la PDP-7). Tanto el BCPL como el B son lenguajes "sin tipos", donde el único tipo de datos es la palabra de la máquina (de modo muy parecido a como ocurre en un lenguaje ensamblador o de máquina). Sin embargo, el lenguaje C es un lenguaje de tipos (o "tipificado"), donde los objetos de datos fundamentales son caracteres, enteros y números de punto flotante. Pero no es considerablemente tipificado (en el sentido de Pascal y Ada); por lo general es muy sencillo realizar la conver-

* UNIX es una marca registrada de Bell Laboratories.

sión de un tipo de datos a otro, aunque el C no permite la conversión de todos los tipos a todos los otros tipos de datos. C también tiene recursos para crear tipos de datos "compuestos" a través del uso de apuntadores, arreglos, estructuras y funciones. Parte de la flexibilidad del lenguaje la demuestra el hecho de que, a pesar de que es un lenguaje estructurado en bloques, la mayor parte del sistema operativo UNIX, el compilador C mismo y la mayor parte de los otros programas de aplicaciones de UNIX se escriben en C.

El C es un lenguaje relativamente de bajo nivel, en el sentido de que maneja muchos de los mismos tipos de objetos que el lenguaje ensamblador (caracteres, números y direcciones). El C realiza operaciones que funcionan directamente con estos tipos de datos. No maneja operaciones de tipos de datos compuestos, como cadenas de caracteres, listas y arreglos completos, aunque estas operaciones son relativamente fáciles de conjuntar a partir de las operaciones más simples que se aceptan. El C mismo no tiene ningún recurso de asignación de almacenamiento que no sea la declaración estática y la capacidad de declarar variables "locales" en funciones, aunque en la mayor parte de las implementaciones se proporciona la asignación de almacenamiento dinámica. La entrada y la salida son realizadas por medio de funciones definidas por la implementación y no por instrucciones del lenguaje. El C no cuenta con constructos complejos de flujo de control (como corrutinas u operaciones en paralelo), sino solamente con constructos directos como pruebas, ciclos, agrupamientos y subprogramas.

Las instrucciones del lenguaje C se terminan con un punto y coma (en contraste con lo que ocurre en el Pascal, donde las instrucciones se separan con puntos y comas). En términos de estructuras de control, el C proporciona instrucciones de repetición con la prueba de control al principio (*for*, *while*) y al final (*do*); toma de decisiones (*if*); y la selección de un elemento entre un conjunto de posibles casos (*switch*)¹⁷. Las formas de estas instrucciones se muestran en la figura 58-17.

El término "stmt" de esta figura es una instrucción única cualquiera del C, y "expr" y "condition" son expresiones por evaluar. Una instrucción *block*, que es un conjunto de una o más instrucciones individuales encerradas entre llaves ("{}"), que agrupan operadores similares a *begin* y *end* de Pascal), puede utilizarse en lugar de una sola instrucción siempre que se desee ejecutar más de una instrucción (como el contenido de un ciclo, por ejemplo). "Case selector" es la especificación de un valor constante y una serie de instrucciones del C por ejecutar, como puede observarse en la figura 58-18.

```
part = record
```

```
    partno, amcunt : integer;
    name : packed array [1..9] of char;
    price : real
```

```
end
```

Fig. 58-15. Declaraciones de registros en Pascal.

```
type item = ^node;
{las variables de este tipo apuntan a elementos del tipo nodo}

var widget, gizmo : item;
```

Fig. 58-16. Declaraciones de variables de apuntador en Pascal.

```
for (expr1; expr2; expr3)
    stmt

while (condition)
    stmt

do
    stmt
while (condition);

if (condition)
    stmt1
else
    stmt2

switch (expr)
{
    case selector_1
    case selector_2
    .
    case selector_n
}
```

Fig. 58-17. Instrucciones de control en lenguaje C.

```
case          const:
              stmt1;
              stmt2;
              .
              stmtn;
```

Fig. 58-18. Selector de casos en lenguaje C.

El C tiene capacidad de definición de macroinstrucciones mediante la instrucción *# define*. Esta instrucción tiene la sintaxis que sigue:

```
# define identifier definition .string
```

donde "identifier" es un identificador legal cualquiera del C y "definition string" es la cadena de caracteres que se sustituirá por "identifier" en cualquier parte en que ocurra (desde el punto de la definición en adelante) del programa. La forma más simple de definición de macroinstrucciones es algo como:

```
# define TRUE 1
```

que tiene el efecto de asociar el nombre TRUE (verdadero) con el valor 1 (esto se parece mucho a la característica *const* del Pascal). Es posible definir macroinstrucciones que tomen argumentos, como en:

```
# define swap (A,B){x=A; A=B; B=x;}
```

que intercambia el contenido de los argumentos A y B. La expansión de las macroinstrucciones se realiza antes de la compilación real de una instrucción; la referencia:

```
swap(alpha, gamma)
```

se expandirá en la instrucción de bloque:

```
{x = alpha; alpha = gamma; gamma = x;}
```

que se compilará entonces.

La entrada y la salida no son parte integral del lenguaje C, sino que se manejan a través de llamadas de subrutinas a rutinas específicas de la instalación. El compilador C no conoce directamente ni los nombres de las funciones I/O ni su sintaxis.

En la figura 58-19 se presenta el programa ordenador de burbuja escrito en lenguaje C. Esta función se llama con dos argumentos: el arreglo por ordenar (que se supone es un arreglo de enteros) y el tamaño del arreglo.

Los arreglos de C se basan en cero; por tanto, un arreglo de tamaño 10 tiene índices del 0 al 9.

Ada. Ada (cuyo nombre se debe a Ada Augusta, Condesa de Lovelace, socia de Charles Babbage y considerada la primera programadora del mundo) es una marca registrada del U.S. Department of Defense. El documento definitivo del lenguaje de programación Ada está disponible como norma militar¹⁸. El U.S. Department of Defense publicó una serie de documentos de requisitos del lenguaje, culminando con *STEELMAN*¹⁹. Después varios contratistas diseñaron lenguajes de programación según estos requisitos (todos los lenguajes competidores se basaron más o menos en el lenguaje Pascal).

Ada difiere de otros lenguajes de programación en que abarca un intervalo mucho más amplio del ciclo de vida del software de la computadora, en particular la actividad de diseño del sistema de software. Ada amplía aún más su campo de acción al tener un entorno operativo definido de manera específica (sistema del tiempo de ejecución)²⁰.

La ventaja del Pascal sobre sus predecesores fue principalmente la incorporación de apoyo para la programación estructurada en el flujo de control del programa y un método coherente de estructuración de datos. Ada incorporó esas características del Pascal y agregó otras para apoyar la modularidad, específicamente paquetes y tareas^{21,22}.

Paquetes. Los paquetes son conjuntos de unidades relacionadas lógicamente como estructuras de datos especiales y sus operaciones asociadas o una familia de subrutinas. Los paquetes pueden tener porciones de detalles de implementación ocultas al usuario, y una parte pública que indica los detalles de enlace necesarios para utilizar el paquete. Los paquetes pueden ser "genéricos" o parametrizados por los tipos de datos

que se necesitan en una aplicación dada. Por ejemplo, podría haber un paquete de ordenamiento genérico que se utilizara para ordenar cualquier tipo de datos (esto es semejante a los recursos de macroinstrucciones de que se dispone en algunos otros sistemas de lenguajes).

El programa ordenador de burbuja de muestra se presenta en la figura 58-20. En él se utiliza un paquete provisto por el sistema, conocido como texto, que contiene todas las rutinas que se necesitan para la entrada y la salida, donde el medio externo es para uso del operador; es decir, es texto (la representación interna suele ser binaria e ilegible).

Un concepto que se maneja en Ada y que por lo general no se incluye en otros lenguajes de programación populares es la "sobrecarga", donde nombres y símbolos pueden representar más de una cosa. El sistema Ada resuelve el significado que se pretende dar en el contexto. En el programa de muestra, el procedimiento "put" se sobrecarga como un procedimiento para imprimir enteros (el primer uso) y como un procedimiento para imprimir cadenas de caracteres (el segundo uso, donde la cadena de caracteres "coma-espacio-espacio" se utiliza para separar los números en la salida). El procedimiento "write" y los operadores aritméticos se sobrecargan en Pascal, pero en Ada los programadores pueden definir los símbolos "+" y "*" como la adición y la multiplicación de matrices.

Tareas. Las tareas (tasks) son unidades para especificar secuencias de acciones que se pueden ejecutar en paralelo con otras unidades similares. Las tareas en paralelo pueden sincronizarse y pasar información por medio de una característica del lenguaje Ada llamada cita de tareas. Las tareas en paralelo se pueden ejecutar en un sistema de procesadores múltiples, con cada tarea en una computadora aparte o en un sistema de un solo procesador y donde las tareas toman turnos de ejecución en segmentos de tiempo.

El procedimiento de ordenamiento de burbuja en Ada se muestra en la figura 58-20. Obsérvese que los comentarios comienzan con dos guiones y terminan con el final de la línea (lo que es mucho menos propenso a error que los convencionalismos de comentarios "{...}" del Pascal o bien "/*...*/" del lenguaje C, donde la ausencia del delimitador del final del comentario puede hacer que se pierda una gran parte del programa, a menudo sin indicación del error). Las estructuras de control del lenguaje Ada son más regulares que las del Pascal. Por ejemplo, en Pascal se escribe:

```
for i: = j to k do
  write (A[i])
```

si solamente existe una instrucción dentro del cuerpo del ciclo, pero:

```
for i: = j to k do
  begin
    read (A[i]);
    write(A[i])
  end
```

```

#define TRUE 1           /*valor verdadero booleano*/
#define FALSE 0          /*valor falso booleano*/

bubble(a, size)          /*encabezado de función*/
    int size, a[ ];       /*definiciones del tipo de argumento*/
    {
        int swapped, i, temp; /*definiciones de (auto)variables locales*/

        do
        {
            swapped = FALSE; /*se supone que no se necesita hacer intercambios*/
            for (i = 0; i <= size - 2; i + +)
                if (a[i] > a[i + 1]) /*¿hay dos elementos fuera de orden?*/
                {
                    temp = a[i]; /*entonces se deben intercambiar*/
                    a[i] = a[i + 1];
                    a[i + 1] = temp;
                    swapped = TRUE; /*y registrar el hecho*/
                } /*fin de if*/
        } while (swapped); /*continuar en tanto se haya hecho un cambio*/
    } /*fin de bubble*/

```

Fig. 58-19. Programa ordenador de burbuja en lenguaje C.

puesto que las dos instrucciones contenidas en el ciclo no necesitan agruparse. En Ada, por otro lado, se escribe:

```

for i in j..k loop
    put (A(i));
end loop;

```

y:

```

for i in j..k loop
    get (A(i));
    put(A(i));
end loop

```

Obsérvense asimismo las diferencias en el uso del punto y coma; en Pascal este signo *separa* instrucciones, pero en Ada *pone fin* a instrucciones. La elección de Ada es mucho más simple de explicar y de seguir, de manera que es menos propensa a error.

SNOBOL4. El lenguaje SNOBOL4 proviene de los Bell Laboratories y ha estado disponible desde mediados de la década de 1960. Como el nombre indica, el lenguaje ha tenido algunas versiones hasta madurar a su estado presente^{23,24}.

Aunque el SNOBOL4 contiene tipos de datos, operaciones y estructuras de control adecuadas para un lenguaje de programación de uso general, sus puntos fuertes y áreas de aplicación radican en el procesamiento de cadenas de caracteres. Una cadena de caracteres es simplemente una secuencia de letras, cifras, signos de puntuación y otros símbolos; por ejemplo, las palabras, enunciados, libros, programas de computadora y líneas de un directorio de teléfonos son cadenas de caracteres. El SNOBOL4 realiza direc-

tamente varias operaciones con cadenas de caracteres, como la concatenación (combinación de dos cadenas para formar una más larga) y el pareamiento de subcadenas (determinación de la incidencia de una cadena dentro de otra, como hallar una palabra importante en un libro).

La característica más interesante y poderosa del SNOBOL4 es el "patrón" para parear subcadenas de cadenas más largas. Con los patrones se pueden componer programas poderosos muy breves, por ejemplo para conversión de programas de computadora escritos en un dialecto del FORTRAN de un fabricante al de otro, modificación de estructuras de registros de un archivo de datos, o manejo de álgebra simbólica.

En algunos programas muy complicados, como los empleados para el procesamiento de idiomas naturales (p. ej., inglés o francés) o la manipulación de la estructura de compuestos químicos, el SNOBOL4 puede realizar el manejo de sus caracteres subyacentes, lo que facilita de manera considerable cuando menos el tratamiento de ese aspecto del problema.

En la figura 58-21 se presenta el programa ordenador de burbujas en SNOBOL4.

Este programa no utiliza ninguna de las características distintivas del SNOBOL4, sino que indica la sintaxis básica del lenguaje.

Cada instrucción del programa está en una línea aparte (originalmente, al igual que el FORTRAN, SNOBOL4 servía para sistemas orientados a las tarjetas perforadas) y comienza con un rótulo opcional (GOAGAIN, TEST, SWAP y NOSWAP). Se hace referencia a los rótulos por medio de la sección go-to condicional, que es la parte final (y opcional) de cada instrucción. Por ejemplo, la instrucción:

```
TEST GT(A(I),A(I+1)) :S(SWAP)F(NOSWAP)
```

```

with text_io; use text_io; use integer_io;
procedure sorter is

  SWAPPED : BOOLEAN;
  I, TEMP : INTEGER;
  A : array (1..10) of INTEGER;

  for I in A'FIRST..A'LAST loop
    get (A(I));
  end loop;

  loop
    SWAPPED := false;
    for I in A'FIRST..A'LAST - 1 loop
      if A(I) > A(I + 1) then
        TEMP := A(I);
        A(I) := A(I + 1);
        A(I + 1) := TEMP;
        SWAPPED := true;      -- registrar que se necesitó un intercambio.
      end if;
    end loop;
    exit when not SWAPPED;   -- ¿concluyó el ordenamiento?
  end loop;

  for I in A'FIRST..A'LAST loop
    put (A(I));
    put (' ');
  end loop;

end sorter;

```

Fig. 58-20. Programa ordenador de burbuja en Ada.

evalúa la función GT (llamada predicado), la cual, si es verdadera (cuando $A(I)$ es mayor que $A(I + 1)$), "triunfa" y el control pasa a SWAP (la S representa *succeed*, triunfo); por el contrario, si GT no es verdadera, se dice que la instrucción falla y el control pasa a NOSWAP.

El SNOBOL4 no maneja directamente, a través de sus estructuras de control, las ideas de la programación estructurada; la mayor parte del flujo de control se realiza por medio de instrucciones go-to condicionales e incondicionales. Sin embargo, el SNOBOL4 tiene un recurso de subrutinas (que no se analiza aquí). La mayor parte de los programas o subrutinas de SNOBOL4 son lo suficientemente cortos para evitar la plaga del código de espagueti.

Las operaciones de entrada y salida en SNOBOL4 son particularmente simples. Para obtener entrada desde el dispositivo de entrada estándar (un archivo, una lectora de tarjetas o una terminal de usuario), simplemente se evalúa la variable INPUT:

NEWDATA = INPUT :F(ALLGONE)

El control se transfiere a la etiqueta ALLGONE en caso de que se haya llegado al final de la entrada. Para producir salida (a un archivo, a una impresora o una terminal), se asigna un nuevo valor a la variable OUTPUT:

OUTPUT = ANSWER

Un ejemplo de pareamiento de patrones del SNOBOL4 se ilustra en la figura 58-22. Este segmento de programa convierte una palabra del inglés a jergonza de niños (la subcadena consonante inicial de una palabra se cambia del frente de la palabra al final y va seguida de "AY"; si la palabra comienza con una vocal, simplemente se suma "WAY" al final; y obsérvese que "QU" es una cadena consonante para estos fines). El lector deberá intentar seguir esto y verificar que DOG se convierte en OGDAY, STREET en EETSTRAY, APPLE en APPLEWAY, SQUIRT en IRTSQUAY.

Al igual que muchos de los otros lenguajes de programación, se observa que el SNOBOL4 tiene el tipo usual de instrucción de asignación:

variable = value

donde el valor, value, del lado derecho es el nuevo valor de la variable, variable, del lado izquierdo. El SNOBOL4 también tiene:

variable pattern = value

donde value es el nuevo valor de la subcadena de variables pareada por pattern. En la figura 58-22 se cambió la subcadena 'QU' de PWORD por 'Q#'; el tramo inicial de consonantes se cambió por una serie nula (no hubo un valor del lado derecho); y el símbolo '#' de PWORD se cambió por 'U'.

```
* ORDENAR EL ARREGLO DE NÚMEROS, A.
  SWAPPED = 'FALSE'
  GOAGAIN I = 1
  TEST GT (A<I>, A<I + 1>) :S(SWAP)F(NOSWAP)
  SWAP TEMP = A<I>
  A<I> = A<I + 1>
  A<I + 1> = TEMP
  SWAPPED = 'TRUE'
  NOSWAP
  I = I + 1
  A<I + 1> :S(TEST)
  EQ(SWAPPED, 'TRUE') :S(GOAGAIN)
```

Fig. 58-21. Programa ordenador de burbuja en SNOBOL4.

La instrucción:

variable₁ pattern . variable₂ = value

sustituye la subcadena pareada de variable₁ por value y guarda la subcadena pareada original en variable₂.

El SNOBOL4 tiene un rico repertorio de patrones; aquí solamente se ha presentado una pequeña muestra:

- 'QU' se para con la cadena 'QU'.
- 'A'|'E'|'I'|'O'|'U' se para con una vocal cualquiera.
- POS(0) 'A'|'E'|'I'|'O'|'U' se para con una vocal cualquiera al inicio de una cadena.
- SPAN ('#BCDFGHJKLMNPQRSTVWXYZ') se para con la cadena más larga posible de consonantes y símbolos #.

Se pueden construir patrones encima de otros mediante el uso de la concatenación y la alternación ("|") para obtener casi cualquier pareamiento de patrones que se pueda imaginar.

APL. El APL (*a programming language*, un lenguaje de programación)²⁵ fue inventado por el Dr. Kenneth

Iverson durante su estadía en Harvard University. En 1966 apareció la primera versión experimental de este lenguaje para uso interno en IBM, y desde entonces se ha convertido en producto oficial de esta empresa. Su desarrollo ha sido promovido más vigorosamente por la I. P. Sharp Company de Toronto, Canadá. La compañía encabezó la producción y promoción de una versión denominada APL + ("APL Plus"), cuyas extensiones principales son:

1. Capacidad de manipular archivos de datos.
2. Capacidad de formateo de informes, que debe mucho al lenguaje FORTRAN y poco al APL.

El APL fue concebido como herramienta de trabajo para matemáticos expertos en estadística e ingenieros. Las construcciones y notación son matemáticas. Una vasta selección de operaciones primitivas se concentra en las técnicas de solución de interés para estos profesionales; por ejemplo, el operador "de dominó", que especifica la inversión de matrices. El APL se apoya en la ejecución interpretativa del programa fuente, el cual, al igual que el BASIC, es fundamental para alentar que la programación sea realizada por el usuario final en vez de confiar en personas cuya especialización sea la programación en sí y no el área de aplicación. Como sucede con el BASIC, la idea de Iverson consistía en proporcionar a los usuarios una interfaz flexible con el poder de cómputo; flexible con respecto a sus capacidades, los tipos de problemas solubles, las expansiones y contracciones de las soluciones a esos problemas, y su implementación. La clave para esto último es la ejecución interpretativa, que hace posible que los usuarios interrumpan programas en ejecución e impriman variables del programa, cambien sus valores y modifiquen el programa en el curso de la ejecución, para reiniciar después la ejecución desde el punto de la interrupción.

Como sucede con otros lenguajes diseñados y utilizados inicialmente con un fin específico, el APL se ha

```
* CÓDIGO PARA TRADUCIR EWORD A PWORD
  EWORD POS(0) 'A'|'E'|'I'|'O'|'U' :F(CONS)
*
*
* EWORD COMIENZA CON UNA VOCAL.
  PWORD = EWORD 'WAY' : (DONE)
*
* EWORD COMIENZA CON UNA CONSONANTE.
  CONS PWORD = EWORD
* CAMBIAR EL PRIMER QU (SI LO HAY) POR Q#.
  PWORD 'QU' = 'Q#'
* RETIRAR LA CADENA CONSONANTE INICIAL DE PWORD.
  Y GUARDARLA EN INIT.
  PWORD SPAN('#BCDFGHJKLMNPQRSTVWXYZ') . INIT =
* AHORA RECONSTRUIR PWORD COMO SE DESEA.
  PWORD = PWORD INIT 'AY'
* RESTITUIR LA "U" SI SE HA MODIFICADO.
  PWORD '#' = 'U'
  DONE
```

Fig. 58-22. Procedimiento escrito con el estilo utilizado por SNOBOL4.


```

▽ BUBBLE
[1] 'VECTOR DE ENTRADA DE NÚMEROS POR ORDENAR'
[2] VEC ← 0
[3] RES ← APLSORT VEC
[4] 'SALIDA DE APLSORT'
[5] RES
[6] FIN ← BUBBLESORT VEC
[7] 'SALIDA DE BUBBLESORT'
[8] FIN
[9] 'END'

▽ RR ← BUBBLESORT A
[1] AGAIN : SWAP ← 0
[2] I ← 1
[3] LOOP:
[4] → (I = ρA)/EXIT
[5] → (A[I] < A[I + 1])/TEST
[6] TEMP ← A[I]
[7] A[I] ← A[I + 1]
[8] A[I + 1] ← TEMP
[9] SWAP ← 1
[10] TEST : I ← I + 1
[11] → LOOP
[12] EXIT: → ((SWAP = 1), SWAP = 0)/AGAIN, FINISH
[13] 'ERROR': → 0
[14] FINISH: 'FINAL DE BUBBLESORT'
[15] RR ← A

▽ R ← APLSORT NUMS
[1] R ← NUMS[ΔNUMS]

```

Fig. 58-23. Cómo ordenar un arreglo en APL.

vuelto popular y su uso se ha extendido a problemas ajenos al dominio original. En forma específica, el APL se está utilizando esencialmente para el procesamiento de datos empresariales. Esta popularidad se debe a dos atributos del lenguaje:

1. Su flexibilidad para manipular matrices o tablas ("asignación dinámica de tamaño" y manipulación de arreglos de datos multidimensionales) ha alentado el uso del APL en el procesamiento de datos de planificación (p. ej., ventas por año, por producto, por región y por sucursal) y clases similares de problemas de arreglos grandes.
2. Constituye un alivio para los ejecutivos, presionados por la insistencia de los departamentos de sistemas de información para que "realmente comprendan y documenten los requisitos antes de realizar la codificación", lo que en consecuencia alarga el ciclo de desarrollo de los sistemas nuevos. Es posible establecer sistemas óptimos en una décima parte del tiempo que requieren los métodos "clásicos", y el sistema inicial se puede emplear como piloto para guiar el desarrollo de un sistema final. Si la versión del APL es demasiado lenta, se puede utilizar como prototipo de trabajo o versión aproximada de la versión final.

En la figura 58-23 se muestran tres funciones del APL:

1. BUBBLE es un programa principal que lee los datos, llama a las dos funciones de ordenamiento e imprime los datos ordenados.
2. BUBBLESORT es la codificación APL que corresponde directamente a los ejemplos de programación que se están presentando, pero en este ejemplo no se aprovechan las capacidades del APL.
3. APLSORT es la forma en que se escribiría el ordenador de burbuja en APL, aprovechando la función de "escalación ascendente", que determina la lista de subíndices de un arreglo en orden creciente conforme a los valores del arreglo.

LISP. El lenguaje LISP (*list processor*, procesador de listas) fue diseñado en MIT como lenguaje para el procesamiento de símbolos²⁶. LISP cuenta con recursos para la manipulación de datos no numéricos y numéricos; la facilidad con que se pueden manipular datos simbólicos hace del LISP el lenguaje preferido para muchas aplicaciones en que se utiliza este tipo de datos. Otra característica interesante del LISP es el hecho de que toda la información, tanto programas como datos, se representa de la misma manera, haciendo posibles acciones comunes, como la creación y evaluación dinámicas de funciones.

Lenguajes ensambladores. El nombre "lenguaje ensamblador" no designa un lenguaje de programación estandarizado individual, sino la forma simbólica para especificar programas escritos en lenguaje de máquina para una computadora dada cualquiera. Como los formatos de instrucciones y datos de diversos modelos de computadoras difieren ampliamente, los lenguajes ensambladores también varían. Pero al igual que en otros aspectos de este dominio, aunque los detalles nunca son los mismos, los conceptos subyacentes casi no varían.

Los lenguajes ensambladores proporcionan los siguientes servicios, que ahorran trabajo a los programadores:

1. Símbolos mnemotécnicos de las operaciones de la máquina (como "L" para "registro de carga" y "MVC" para "mover caracteres").
2. Nombres simbólicos ("etiquetas") para las localidades de instrucciones y datos.
3. Nombres simbólicos para constantes.
4. Expresiones en que intervienen nombres y constantes simbólicos.
5. Inicialización de datos en formas legibles para el operador (es decir, constantes decimales en vez de patrones binarios).
6. Comentarios en líneas respectivas separadas y comentarios anexos a cada instrucción.

La programación en lenguaje ensamblador se ilustrará en términos del lenguaje ensamblador básico (también conocido como BAL) de los sistemas 360/370 de IBM para expresar el procedimiento de ordenamiento de burbuja²⁷. La sintaxis de BAL es como sigue:

```

* 10 SWAPPD = .FALSE.
L10 MVI SWAPPD, colocar el byte SWAPPD en 0.
* DO 100 I = FIRST, LAST-1
  L 1, FIRST  colocar el valor FIRST en el registro general 1.
  ST 1, I  almacenar el contenido del registro general 1 en I.
DO100 L 1, I
  C 1, LAST  comparar el contenido del registro 1 en LAST.
  BGT DO100X ir a DO100X si I es mayor o igual que LAST.
* IF (.NOT. (A(I) .GT. A(I + 1))) GOTO 100
  SLL 1, 2  desplazar el registro general dos bits a la izquierda.
  LE 0, A(1)  cargar el I-ésimo elemento de A en el registro de punto flotante 0.
  CE 0, A + 4(1)  comparar el registro de punto flotante 0 con el
                  I + 1-ésimo elemento de A.
  BLE L100 ir a L100 si la comparación es "menor o igual que".
* TEMP = A(I)
  STE 0, TEMP  almacenar el registro de punto flotante 0 en TEMP.
* A(I) = A(I + 1)
  LE 0, A + 4(1)
  STE 0, A(1)
* A(I + 1) = TEMP
  LE 0, TEMP
  STE 0, A + 4(1)
* SWAPPD = .TRUE.
  MVI SWAPPD, 1
* 100 CONTINUE
L100 B DO100 ir a la etiqueta DO100.
* IF (SWAPPD) GOTO 10
DO100X CLI SWAPPD, 1  comparar SWAPPD con 1.
  BE L10 ir a L10 si la comparación fue "igual a".

```

Fig. 58-24. Programa ordenador de burbuja en lenguaje ensamblador.

1. Las líneas de comentarios contienen un asterisco ("*") en la posición uno.
2. Las etiquetas, si las hay, comienzan en la posición uno.
3. El código de operación ("opcode" o código op) va después de la etiqueta, precedido de uno o más espacios en blanco. Si el código op es un símbolo mnemotécnico de una instrucción de la máquina, entonces la línea actual corresponde a una sola instrucción de la máquina; de lo contrario, la línea se utiliza a fin de reservar almacenamiento para variables del programa (posiblemente inicializándolas) o proporcionar directrices al ensamblador.
4. Después del código op, separado por uno o más espacios en blanco, está el campo de operandos, que consta de una lista de especificadores de variables del programa, registros de la máquina y parámetros que influyen en la interpretación del código de operación.
5. Después de la lista de operandos, separada por uno o más espacios en blanco, está el campo de comentarios, que consta de cualquier anotación que el programador desee hacer acerca de la instrucción (los comentarios son opcionales).

En la figura 58-24 se usa el código de la subrutina ordenadora de burbuja en lenguaje FORTRAN para anotar instrucciones en lenguaje ensamblador.

Como el lector puede advertir en el ejemplo, el lenguaje ensamblador es muy difícil de leer y escribir y carece de las características que se dan por un hecho en los lenguajes de alto nivel, como instrucciones IF, ciclos y subrutinas. Todo se debería hacer como una secuencia de pasos pequeños; cada paso es muy simple, pero existen tantos de ellos que los programas se vuelven abrumadoramente complicados. Muchos observadores han llamado la atención sobre el hecho de que la productividad de los programadores, medida en términos de líneas de código producidas por día, es independiente del lenguaje de programación elegido (los números que se informan son del orden de 7 a 50 líneas por día, donde la variación depende únicamente de la capacidad individual). A pesar de esto, se han dado varias razones para el uso del lenguaje ensamblador:

1. Los compiladores de los lenguajes de alto nivel no son tan eficientes como los programadores del lenguaje ensamblador. El código generado por el compilador es más largo y más lento. (Este argumento se vuelve cada vez menos válido conforme avanza la tecnología de los compiladores. El código generado por el compilador tiene mayor probabilidad de ser correcto y fácil de mantener.)
2. Las máquinas nuevas pueden no tener compiladores en operación, y los ensambladores son

más fáciles de construir que los compiladores. (Este argumento es abrumador para el programador de aplicaciones, pero es válido para muy pocas máquinas y sólo por un breve periodo de tiempo).

58.5 SIMULACIÓN DE SISTEMAS

Los sistemas en que intervienen modelos de simulación representan un área de aplicación de las computadoras de interés e importancia crecientes. Expresada en forma simple, la simulación implica una metodología para el estudio de la dinámica de un sistema o, en forma alternativa, el estudio de la interrelación de las componentes que conforman el sistema. El **sistema** puede definirse como aquel que consta del conjunto de partes que están organizadas funcionalmente para conformar el todo. Por ejemplo, un puerto consta de muelles, remolcadores, barcos, canales, vías navegables, etc. Todas estas componentes se combinan para conformar un sistema que se puede definir como un puerto. De hecho, sin las componentes el puerto no es más que una simple caleta. Es la interrelación de las componentes del puerto lo que interesa a quien genera el modelo (el modelador).

Un modelo es una representación del sistema. Describe aquellas componentes que lo constituyen con el suficiente detalle para permitir estudiar el comportamiento del sistema en intervalos válidos de restricciones operacionales y, lo que es más importante, hacer predicciones válidas acerca del comportamiento del sistema o, si se requiere, seleccionar planes de acción alternativos. Por tanto, es importante que se incluyan en el modelo las componentes que tengan un efecto significativo en el comportamiento del sistema. Son de particular interés las componentes a las cuales es más sensible el modelo, aunque el modelador de la simulación siempre se enfrenta a la justificación (o racionalización) de la inclusión (o exclusión) y del comportamiento de cada componente incluida en el modelo, sin importar su impacto²⁸. Dos tipos de actividades concurren en el modelo típico: actividades endógenas (parámetros) que describen las actividades que ocurren en el sistema, y actividades exógenas (entrada) que describen aquellas actividades que tienen un efecto en el sistema.

Si bien con un modelo se pretende reproducir hasta cierto punto un proceso real, el modelo de simulación no es un intento de imitar el mundo real. Algunas de las razones principales de esta filosofía son: no es raro ver que el funcionamiento del sistema depende sólo de relativamente pocas componentes; a menudo es poco práctico crear un modelo muy detallado, en contraste con un prototipo del sistema; algunas veces las interrelaciones contenidas en el sistema no son claras o escapan al campo de acción del análisis; el reunir grandes cantidades de datos empíricos, que se utilizan para determinar parámetros, valores de entrada o interrelaciones, puede consumir tiempo, ser costoso o poco práctico; las restricciones de costo y tiempo suelen limitar el alcance del modelo.

Se ha dicho²⁹ que "la simulación por computadora es el tribunal de última instancia". Si hubiese alguna otra manera de analizar el sistema, debería utilizarse. Desafortunadamente algunas veces esto se interpreta como que siempre existe un tribunal de última instancia, aunque por desgracia algunos problemas no se pueden analizar, ya que no es plausible un análisis de sistemas en profundidad.

Los modelos generados por computadora tienen mayor probabilidad de ser estocásticos que deterministas. Un proceso determinista es aquel en el cual todos y cada uno de los valores de entrada producen algún valor de salida en forma reproducible. Una ecuación matemática como $a = b + 4$ representa un proceso determinista, ya que para cada valor de entrada de b el valor de a siempre se determina de manera única sin ninguna irregularidad. En contraste, un proceso estocástico implica el concepto de la aleatoriedad. Dados suficientes datos empíricos, se pueden hacer predicciones acerca del comportamiento de un modelo estocástico suponiendo que el análisis del sistema ha determinado las interrelaciones entre las componentes del modelo. Los modelos estocásticos por lo general se componen de subsistemas más probablemente estocásticos. Es la interacción de los subsistemas de un sistema lo que interesa al modelador.

Los modelos de computadora también se pueden clasificar como modelos de eventos discretos o continuos²⁸. Los modelos continuos se utilizan para representar la dinámica de los procesos, y más específicamente el comportamiento de dispositivos o fenómenos físicos continuos que se representan mejor por medio de técnicas matemáticas como ecuaciones diferenciales o integración numérica. En lugar de intentar reproducir el comportamiento continuo de un sistema, los modelos de eventos discretos utilizan intervalos de tiempo discretos en los cuales se observa el comportamiento del sistema. La suposición implícita es que la unidad de tiempo seleccionada es lo suficientemente corta para hacer posible la captura de estadísticas de desempeño apropiadas, sin la posibilidad de que la varianza de desempeño extrema observada tenga un periodo de tiempo menor.

Algunas veces un modelo de eventos discretos puede tener un modelo continuo como uno de sus subsistemas. Por ejemplo, esto ocurriría cuando un subsistema se pudiera describir en forma adecuada por medio de las leyes de la física. Por desgracia, la inclusión de un subsistema continuo dentro de un modelo de eventos discretos no siempre es factible si se utiliza un lenguaje de simulación de eventos discretos, como el GPSS o Simscript 11.5²⁸⁻³⁰. Sin embargo, se deberá observar que no toda la simulación de eventos discretos se realiza mediante el uso de un lenguaje de simulación de eventos discretos de uso especial. Los modelos se pueden implementar en FORTRAN, PL/I, Pascal o Ada. Aun el BASIC y el COBOL son posibles lenguajes. No obstante, para tratar la naturaleza estocástica del modelo, se deberá disponer de un generador de números pseudoaleatorios.

Un estudio de simulación tiene varias fases, y el éxito en cada una de ellas es esencial si se espera ob-

tener un buen resultado. Primero viene la fase de análisis del sistema, para definir el modelo mismo. Esto comprende la determinación de las componentes que se incluirán en el modelo, la observación de interrelaciones y la recolección de datos empíricos que se usarán como guía en el diseño del modelo y como valores de parámetros y datos de entrada del modelo. En segundo lugar está la programación de la computadora, a fin de codificar el modelo para su procesamiento. Esto incluye la selección de un lenguaje de programación adecuado.

En la tercera fase interviene la depuración del programa de computadora que representa el modelo. En la modelación, esto implica varios pasos de verificación. Este último término significa determinar si el programa produce una salida consistente con la forma en que se espera que se comporte el modelo. Una técnica común para realizar este proceso es la "formación de cajas"²⁹, en la cual una porción del programa se "mete en una caja" y se analizan la entrada y la salida de la porción del programa contenida en la caja. Como aquí se está tratando un procesamiento de tipo no procesal en el que pueden ocurrir varios eventos de manera concurrente, la verificación del programa no sólo es más difícil de lo usual sino también más esencial.

La cuarta fase es la validación. En esta fase se busca incrementar la confianza del usuario en el modelo; esto es, ¿con qué eficiencia corresponde la salida del modelo al comportamiento esperado del sistema? Al igual que las tres primeras fases, la validación es esencial antes de que se puedan hacer predicciones o elegir alternativas. La quinta fase implica la iteración. El procesamiento iterativo del modelo se realiza por tres razones:

1. Para continuar el proceso de validación.
2. Para determinar la sensibilidad del modelo a sus diversas componentes, lo que puede sugerir otras estrategias o la adición o supresión de componente del modelo antes de que se realicen corridas subsiguientes en la computadora.
3. Determinar si el modelo ha alcanzado una condición de estado estable (¿se debe la varianza de la salida a procesos estocásticos implicados en el modelo [una condición de estado estable], o se debe a fluctuaciones que se eliminarían si el modelo se ejecutara por un periodo de tiempo más largo o si se ejecutara en n interacciones más antes de analizar la salida del modelo con el fin de hacer predicciones o elegir alternativas?). El costo, tiempo y esfuerzo implicados en una simulación por computadora óptima no suelen ser en absoluto triviales.

Además de los lenguajes de programación de uso general, se dispone de varios lenguajes de simulación de eventos discretos de uso especial.

58.5.1 GPSS

El GPSS (hacia 1960) se destinó originalmente sólo al uso interno en IBM para el análisis de proyectos de ingeniería^{29,31}. Se lanzó públicamente al mercado en

1961 como GPSS. (Originalmente se conoció como GPS.) GPSS es el acrónimo de General Purpose Simulation System (sistema de simulación de uso general), y actualmente IBM lo designa GPSS-V. Otros fabricantes de computadoras y distribuidores de software apoyan dialectos de GPSS para equipo Burroughs, Honeywell, UNIVAC, Xerox y DEC. GPSS sigue siendo probablemente el lenguaje de simulación de eventos discretos de uso especial más común. Es un lenguaje orientado a los bloques, donde cada bloque del diagrama corresponde a una instrucción de GPSS. Inherente en un procesador GPSS está cuando menos un generador de números pseudoaleatorios y un "reloj" que se incrementa en forma automática para el usuario. El GPSS es un lenguaje orientado a las transacciones. Se generan transacciones que circulan a través del modelo desde un bloque al otro hasta que llegan a su fin. Las estadísticas estándares acumuladas por el GPSS reflejan el efecto acumulativo de todas las transacciones que han pasado por el modelo durante el periodo de tiempo transcurrido. El GPSS describe mejor el efecto global sobre el modelo de todas las transacciones, comparado con un lenguaje orientado a los eventos, en el cual resulta más sencillo capturar estadísticas referentes a cada entidad individual.

Los programas GPSS suelen estar conformados de varios subprogramas; es la interacción entre éstos la que refleja la naturaleza estocástica del sistema representado.

58.5.2 Simscript II.5

El Simscript II.5 (hacia 1961) fue producido por la Rand Corporation³⁰. Tiene sintaxis parecida a la del idioma inglés, una descripción de órdenes de alto nivel en contraste con la sintaxis más orientada al ensamblador de GPSS, y alienta la formación lógica de módulos. El Simscript se concentra en el concepto de evento, que se define como un proceso cualquiera que hace que el sistema cambie. En efecto, un evento es una rutina. A diferencia de lo que ocurre en el GPSS, en el Simscript el tiempo transcurre entre eventos y no dentro de eventos. En un programa Simscript, una entidad pasa de un evento al que sigue conforme avanza el reloj del sistema. Como sucede con el GPSS, las implementaciones del Simscript siempre tienen, cuando menos, un generador de números pseudoaleatorios y un reloj del sistema integrado que se incrementa en forma automática. Si bien es definitivamente posible capturar estadísticas globales concernientes al desempeño del modelo, resulta mucho más sencillo capturar estadísticas referentes a cada evento del Simscript, en comparación con lo que ocurre en el GPSS. El Simscript supera asimismo una de las más grandes debilidades del GPSS, al admitir instrucciones aritméticas como las del FORTRAN en un programa escrito en lenguaje Simscript.

58.5.3 SIMULA

El lenguaje SIMULA (hacia 1960) está basado en el Algol-60; una implementación común es el SIMU-

LA-67³¹. Una de sus características más sobresalientes es que está integrado en un lenguaje anfitrión (Algol), de manera que están disponibles para el usuario todas las capacidades de un lenguaje de programación de uso general. El SIMULA observa el modelo del sistema como un conjunto de procesos en el cual se puede cambiar el estado de un proceso a medida que varía la estructura de datos asociada con ese proceso. SIMULA se alinea más de cerca con el GPSS y su orientación a las transacciones.

58.5.4 GASP

El GASP (hacia 1974) fue desarrollado por Pritsker Associates de West Lafayette, Indiana³². En él se utiliza un método de planificación por eventos similar al del Simscript. Incluye recursos para manejar modelos híbridos, que contienen componentes de eventos discretos y continuos.

58.5.5 SLAM

El SLAM (hacia 1979) es otro lenguaje de simulación de alto nivel que hace posible la simulación de eventos discretos y de eventos continuos. Se ha implementado en el VAX/780, CDC-6000, IBM 360-370, UNIVAC 1108, PRIME 400 y 700, y Harris 550. El SLAM emplea una estructura de red con símbolos especializados de nodos y ramas. Con él se pretende representar un sistema en forma pictórica.

58.6 SISTEMAS DE BASES DE DATOS

Una **base de datos** se define como un conjunto de datos operacionales almacenados, utilizados por los sistemas de aplicaciones de alguna empresa³³. Es, por tanto, un sistema cuyo fin es el de almacenar, conservar y proporcionar información de la base de datos.

Los bloques fundamentales de la base de datos son entidades y relaciones entre entidades. Las entidades son aquellos objetos en los que tiene interés una empresa. (Un ejemplo es la colección de máquinas propiedad de la empresa.) Las entidades se describen por medio de características llamadas atributos. Por ejemplo, la entidad *máquina* puede estar compuesta de nombre, número y ubicación de la máquina.

Las entidades están asociadas entre sí a través de relaciones. Las relaciones pueden ser implícitas o explícitas, dependiendo del modelo que se emplee para diseñar la base de datos. Por ejemplo, la entidad *máquina* puede relacionarse con una entidad que describe piezas que conforman la máquina. La relación entre la máquina y las piezas sería aquella que indica que una máquina está compuesta de piezas.

Una ventaja de los sistemas de bases de datos es la capacidad de modelación que ofrecen. Un sistema de bases de datos puede modelarse de manera que la organización de los datos mismos aporte información sobre la empresa. Actualmente existen tres modelos bien conocidos de sistemas de administración de bases de datos: relacional, celular y jerárquico³³.

Modelo relacional. Se deriva de la teoría matemática de conjuntos. Las relaciones representan entidades y se definen en forma tabular. Cada relación (tabla) está compuesta de atributos (columnas) que describen la relación. Cada renglón de una tabla, llamado *tupla*, representa una ocurrencia de la entidad. Las relaciones entre entidades no se especifican en la base de datos, sino que las especifica el usuario a través del lenguaje de interrogación. La información se recupera de una base de datos relacional mediante el uso de interrogaciones basadas en el álgebra relacional.

Modelo celular de datos. En este modelo las relaciones existentes entre entidades se expresan en forma explícita por medio de relaciones de propiedad. Una entidad puede ser un propietario en muchos conjuntos y ser un miembro de muchos otros.

Modelo jerárquico de datos. Es un caso especial del modelo celular, en el cual una estructura gráfica general se limita a una estructura de árbol.

Un sistema de administración de bases de datos es un conjunto de software diseñado para almacenar, conservar y recuperar información de la base de datos. Tales sistemas contienen recursos de definición de datos que permiten al usuario diseñar y describir la base de datos. Esos recursos se basan en software interno que dirige el almacenamiento físico de los datos. Un sistema de bases de datos ofrece a cada usuario un "panorama" de los datos consistente con sus necesidades particulares. También se dispone de un recurso de manipulación de datos dentro del sistema de bases de datos. Este lenguaje de manipulación permite al usuario anexo y suprimir información de la base de datos y también actualizar y recuperar datos según se necesite. La recuperación y la actualización pueden realizarse en línea con un lenguaje de interrogación no procesal, o a través de llamadas desde programas de aplicación escritos en COBOL, FORTRAN o lenguaje ensamblador.

El uso de un sistema de bases de datos por una empresa ofrece muchas ventajas, tales como el control centralizado de los datos. Muchas áreas funcionales de una empresa emplean los mismos datos. En consecuencia, este tipo de control facilita su compartición. Como la única copia de los datos que existe está almacenada, una sola operación de la base de datos ejecutada con los datos de la base realiza la operación para todos los usuarios. De este modo, no sólo se mantiene en un nivel mínimo el almacenamiento, sino que todos los datos que utiliza la empresa se mantienen actuales y consistentes, lo que garantiza la integridad de los datos. Sin embargo, esta integridad de los datos deberá pagarse con mayores precauciones de seguridad. A través del control centralizado del acceso a los datos se obtiene un nivel de protección en el que sólo aquellas personas que necesiten los datos puedan tener acceso a ellos.

Otra ventaja de los sistemas de bases de datos es que dan independencia física de datos. Existe independencia física de datos entre los programas de aplicación y

la base de datos cuando los cambios de uno no afectan al otro. Un alto grado de esta independencia hace posible que los programas de aplicación de bases de datos sean modificados con mínimo esfuerzo de mantenimiento de la base de datos. La independencia física de datos también hace posible cambiar la base de datos con un mínimo efecto sobre los programas de aplicación.

El futuro de la tecnología de bases de datos es promisorio, pero los simpatizantes se clasifican en dos campos. Existen aquellas personas que se inclinan por el modelo relacional por su sencillez y sus fundamentos teóricos. Sin embargo, el modelo relacional no se ha implementado con buenos resultados para administrar de manera efectiva bases de datos muy grandes. Aquellos que se inclinan por el modelo celular apoyan un modelo que se ha implementado pero que tiene considerable complejidad y poca flexibilidad³⁴.

REFERENCIAS BIBLIOGRÁFICAS

1. E. N. Yourdon, ed., *Classics in Software Engineering*, Yourdon Press, Nueva York.
2. D. E. Knuth, *The Art of Computer Programming*, Vol. 3, *Sorting and Searching*, Addison-Wesley, Reading, MA.
3. A. V. Aho y J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, MA.
4. P. Naur y colaboradores, "Revised Report on the Algorithmic Language Algol-60", *Commun. ACM* 6(1):1-17.
5. A. van Wijngaarten, "Revised Report on the Algorithmic Language Algol-68", *Acta Inf.* 5: 1-236.
6. D. D. McCracken, *A Simplified Guide to Structured COBOL Programming*, Wiley, Nueva York.
7. L. S. Cohn, *Effective Use of ANS COBOL Computer Programming Language*, Wiley, Nueva York.
8. *ANSI X3.9-1966*, USA Standard FORTRAN.
9. *ANSI X3.9-1978*, American National Standard Programming Language FORTRAN.
10. M. Boillot, *Understanding FORTRAN*, 2.^a ed., West, St. Paul, MN.
11. L. Meissner y E. Organick, *FORTRAN 77: Featuring Structured Programming*, Addison-Wesley, Reading, MA.
12. C. T. Fike, *PL/I for Scientific Programmers*, Prentice-Hall, Englewood Cliffs, NJ.
13. J. G. Kemeny y T. E. Kurtz, *BASIC Programming*, Wiley, Nueva York.
14. K. Jensen y N. Wirth, *Pascal User Manual and Report*, 2.^a ed., Springer-Verlag, Nueva York.
15. *Computer Programming Language, Pascal*, IEEE, Los Alamitos, CA.
16. G. M. Schneider, S. W. Weingart y D. M. Perlman, *An Introduction to Programming and Problem Solving with Pascal*, 2.^a ed., Wiley, Nueva York.
17. B. W. Kernighan y D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ.
18. *Ada Programming Language*, MIL-STD-1815.
19. Department of Defense, *STEELMAN Requirements for High Order Computer Programming Language*.
20. Department of Defense, *Requirements for Ada Programming Support Environments-STONEMAN*.
21. I. C. Pyle, *The Ada Programming Language*, Prentice-Hall, Englewood Cliffs, NJ.
22. P. Wegner, *Programming with Ada: An Introduction by Means of Graded Examples*, Prentice-Hall, Englewood Cliffs, NJ.
23. R. Griswold y colaboradores, *The SNOBOL4 Programming Language*, Prentice-Hall, Englewood Cliffs, NJ.
24. R. Griswold y M. Griswold, *A SNOBOL4 Primer*, Prentice-Hall, Englewood Cliffs, NJ.
25. L. Gilman y A. J. Rose, *APL—An Interactive Approach*, Wiley, Nueva York.
26. J. McCarthy y colaboradores, *LISP 1.5 Programmer's Manual*, 2.^a ed., MIT, Cambridge, MA.
27. G. Struble, *Assembly Language Programming; The IBM System 360/370*, 2.^a ed., Addison-Wesley, Reading, MA.
28. J. A. Payne, *Introduction to Simulation*, McGraw-Hill, Nueva York.
29. T. J. Schreiber, *Simulation Using GPSS*, Wiley, Nueva York.
30. T. P. Wyman, *Simulation Modeling: A Guide to Using Simscript*, Wiley, Nueva York.
31. P. A. Bobillier, B. C. Kahan y A. R. Probst, *Simulation with GPSS and GPSS-V*, Prentice-Hall, Englewood Cliffs, NJ.
32. A. A. Pritsker, *The GASP-IV Simulation Language*, Wiley, Nueva York.
33. C. J. Date, *An Introduction to Database Systems*, 2.^a ed., Addison-Wesley, Reading, MA.
34. E. Wong (coordinador) y colaboradores, *Computer World*, p. 78.

CAPÍTULO 59

Organización del procesador central

Melvyn M. Drossman

New York Institute of Technology
Old Westbury, Nueva York

- 59.1 Estructura y función**
- 59.2 Unidad de control**
 - 59.2.1 Sincronización
 - 59.2.2 Registros
 - 59.2.3 Operación
- 59.3 Instrucciones**
- 59.4 Organización de ductos**
- 59.5 Operación de la CPU**
- 59.6 Unidad aritmética y lógica**
 - 59.6.1 Diseño de la ALU
 - 59.6.2 Generador de previsión de acarreo
- 59.7 Algoritmos aritméticos**
 - 59.7.1 Aritmética de punto fijo
 - 59.7.2 Aritmética de punto flotante
 - 59.7.3 Operaciones de cálculo con BCD
- 59.8 Microprogramación**
 - 59.8.1 Operación
 - 59.8.2 Microinstrucciones
 - 59.8.3 Microrrutinas
 - 59.8.4 Microprogramación horizontal y vertical
- 59.9 Microprocesadores de “rebanadas” de bits**

59.1 ESTRUCTURA Y FUNCIÓN

La unidad de procesamiento central (CPU)¹ de una computadora digital es el elemento funcional principal del sistema de computación. Consta de dos subunidades funcionales: la unidad de control (CU) y la unidad aritmética y lógica (ALU). La unidad de control interpreta instrucciones, hace que las otras unidades de la computadora realicen las funciones que se requieran para ejecutar las instrucciones, y sincroniza su operación. La ALU realiza las operaciones de aritmética y lógica que, junto con las instrucciones de entrada y salida, constituyen el conjunto de instrucciones de la computadora. Las instrucciones aritméticas son la suma y la resta y, en algunas computadoras, la multiplicación y la división. Entre las operaciones de lógica se cuentan la comparación, y algunas o todas las operaciones booleanas AND, OR, XOR (OR excluyente) y complementos. También se realizan una variedad de operaciones de desplazamiento. La CPU opera junto con la unidad de memoria principal y dispositivos de entrada y salida. Para explicar la operación de la CPU es necesario describir los vínculos (interfaces) entre la CPU y estos dispositivos.

El sistema de memoria principal consta de la memoria misma, un registro de direcciones de la memoria (MAR), un registro intermedio de la memoria (MBR), también llamado registro de datos de la memoria (MDR), y un control de la memoria, como se muestra en la figura 59-1. Esta memoria contiene N palabras de W bits por palabra. El registro de direcciones de la memoria contiene la dirección de la palabra por introducir. Este registro contiene K bits, donde $2^{**}K = N$,

el número de palabras que hay en la memoria. ($2^{**}K$ representa 2 elevado a la potencia K .) El registro intermedio de la memoria sirve como interfaz entre la memoria y el resto de la computadora. Los datos se leen de la memoria en el MBR y se escriben del MBR en la memoria. El tamaño del MBR es igual al tamaño de palabra de la memoria. El control de la memoria determina si una palabra de datos se transfiere hacia o desde la memoria y cuándo ocurre la transferencia. En la figura 59-1 se utilizan tres entradas: la primera es la línea de lectura/escritura (RW), que es una operación de estado alto para la lectura de la memoria y de estado bajo para la escritura en la memoria; la segunda es la línea de habilitación de la memoria (ME), que indica que está ocurriendo una operación de acceso a la memoria; y la tercera es la entrada del reloj (CI), que proporciona la sincronización con el resto de la computadora.

Las interfaces con los dispositivos de entrada y salida³ son tipificadas por el dispositivo de salida y el controlador de la figura 59-2. El registro de datos (DR) es cargado con L bits de datos vía el ducto de datos por la CPU, que después coloca el registro de protocolo de reconocimiento (HS) de un solo bit en el estado alto empleando la línea de control SHS. Esto hace que el controlador envíe una señal de control "de inicio de la salida" al dispositivo para comenzar la operación de salida. Los datos se transfieren del DR al dispositivo. Cuando se completa la operación de salida, el controlador recoloca el bit del registro HS en el estado bajo. La CPU detecta que se ha completado la operación de salida leyendo el estado del registro HS mediante la línea de control RHS. La línea de entrada

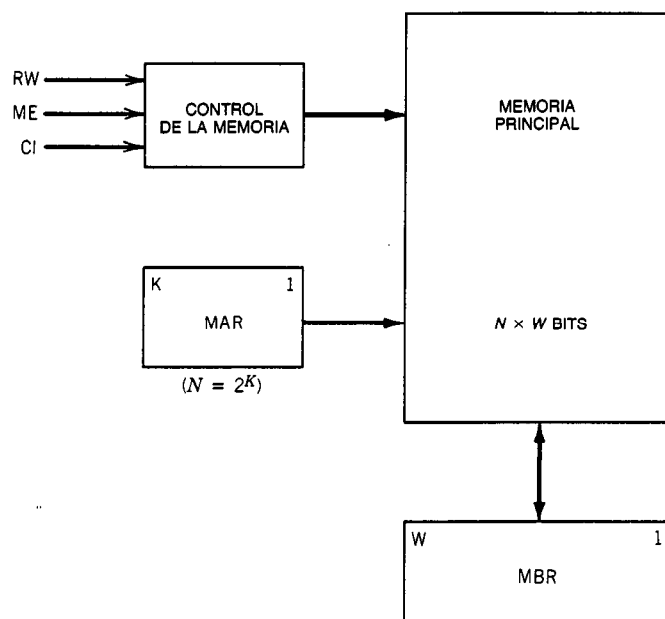


Fig. 59-1. Sistema de memoria principal. RW = lectura y escritura; ME = habilitación de la memoria; CI = entrada del reloj; MAR = registro de direcciones de la memoria; MBR = registro intermedio de la memoria.

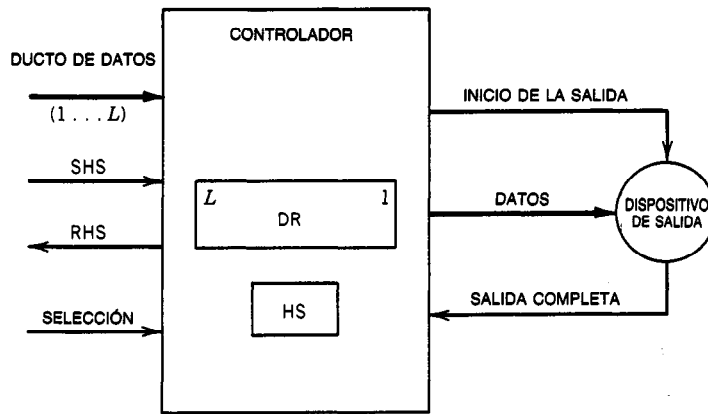


Fig. 59-2. Sistema de salida. DR = registro de datos; HS = registro de protocolo de reconocimiento (*handshaking*); SHS = colocar HS; RHS = leer HS.

SELECT (de selección) se utiliza para seleccionar un controlador específico; si está en el estado bajo, el controlador no responde a ninguna de las líneas de entrada y su línea RHS se encuentra en un estado de alta impedancia.

59.2 UNIDAD DE CONTROL

La unidad de control⁴ se encarga de la operación de la computadora. Captura y decodifica instrucciones, genera las señales de sincronización y establece las series de eventos que ocurren durante la operación de la computadora. Contiene varios registros que almacenan la información que la computadora requiere durante su operación, y controla la transferencia de información entre estos registros y otras unidades de la computadora.

59.2.1 Sincronización

La operación de una computadora típica consiste en la repetición cíclica de una secuencia básica de cuatro fases: la de captura o búsqueda, indirecta, de ejecución y de interrupción. En cada ciclo se procesa una instrucción. La instrucción se busca (captura) en la memoria durante la fase de búsqueda o captura. Durante la fase indirecta se obtiene la dirección real de los datos. La instrucción se ejecuta en la fase de ejecución. La fase de interrupción permite detener el ciclo normal de ejecución de instrucciones en respuesta a una solicitud de interrupción ocasionada por circunstancias especiales. Una vez que se ha completado el procesamiento de la interrupción, se reanuda el ciclo de instrucciones normal con la fase de búsqueda o captura de la siguiente instrucción.

La señal de sincronización para la CPU proviene de un reloj generador, que es un circuito que genera una serie de pulsos a intervalos regulares. El encadenamiento (control secuencial) y la sincronización de la operación de los diversos elementos de la computadora se realizan dirigiendo los pulsos de reloj hacia líneas

diferentes en momentos distintos durante el ciclo de instrucción, mediante una combinación de contador y decodificador o un contador anular (en anillo).

59.2.2 Registros

La operación de la unidad de control se basa en la información almacenada en los siguientes registros, que son parte de la unidad de control:

IC (contador de instrucciones). Se denomina también PC (contador del programa). Este registro es una combinación de registro de almacenamiento y contador que almacena la dirección de la siguiente instrucción por ejecutar.

IR (registro de instrucción). Registro de almacenamiento que se carga con la instrucción capturada de la memoria durante la fase de búsqueda del ciclo de instrucción. A menudo, el registro de instrucciones es reemplazado por varios registros separados que guardan diferentes partes de la instrucción, o sea el código de operación, la dirección de la memoria, el bit indirecto y otros.

Registros de índice. Son por lo general una combinación de registros de almacenamiento y contadores que se utilizan en el direccionamiento indicado, en el cual la dirección efectiva de un dato se determina como la suma del campo de direcciones contenido en la instrucción y el contenido de un registro de índice. Esto permite repetir la misma secuencia de instrucciones y hacer referencia a diferentes localidades de la memoria en cada iteración incrementando el contenido de un registro de índice.

Registro de base. Registros de almacenamiento usados para direccionar el desplazamiento de bases, en los cuales la dirección efectiva de un dato se calcula como la suma del desplazamiento, contenida en el campo de direcciones de la instrucción, y el contenido del registro de base. Esto suele efectuarse para ahorrar espacio

en la instrucción mediante el uso de un campo de desplazamiento que sea más corto de lo que sería el campo de direcciones total.

Los registros que siguen, si bien no son parte de la unidad de control, son parte de la CPU y participan en la operación de la unidad de control:

DR (registro de datos). Registro de almacenamiento que se utiliza para almacenar el operando capturado de la memoria para que lo use la ALU. A veces se utiliza el MBR en vez de un DR.

Acumuladores. Se usan para almacenar operandos y resultados parciales en operaciones de cálculo. Algunas computadoras emplean un solo acumulador, mientras que otras utilizan acumuladores múltiples, de manera que los resultados parciales no tengan que intercambiarse continuamente dentro y fuera de la memoria. En cambio, se usa un acumulador diferente para cada resultado parcial y después se combinan los resultados parciales de diferentes acumuladores. Los acumuladores operan en conjunción con la unidad aritmética y lógica y están directamente conectados a ella. El acumulador suele ser un registro bastante complejo, con una combinación de registro de almacenamiento, registro de desplazamiento y posiblemente también contador.

MQ (multiplicador-divisor). Sirve como extensión del acumulador para realizar operaciones de multiplicación y división en las que intervengan operandos o resultados que tengan dos veces el tamaño de los otros operandos; es decir, el producto de dos números de n bits tiene $2n$ bits de largo. El registro MQ es similar al acumulador, pero un tanto más limitado en su función.

Registros generales. Registros de almacenamiento que forman parte de la CPU. Se utilizan en conjunción con los sistemas de ductos que comúnmente les permiten funcionar como registros de base, registros de índice y acumuladores.

CC (código de condición). Registro de almacenamiento que forma parte de la CPU. Almacena un número de bits pequeño, por lo general cuatro, que reflejan la naturaleza del resultado calculado más recientemente; por ejemplo, si hubo un desbordamiento, o si el resultado fue positivo, negativo o cero. Las instrucciones de ramificación condicional se ramifican o no, dependiendo del valor del código de condición.

59.2.3 Operación

Durante la fase de búsqueda o captura, la instrucción se captura de la localidad de la memoria cuya dirección esté en el contador de instrucciones (IC) y se almacena en el registro de instrucción (IR). El contador de instrucciones se incrementa en el número de palabras en la memoria por instrucción después de que se captura cada instrucción, lo que hace que contenga la dirección de la siguiente localidad de la memoria al principio del

ciclo de búsqueda siguiente. Esto provoca la ejecución secuencial normal de las instrucciones procedentes de localidades consecutivas de la memoria. Si ocurre una operación de ramificación, el IC se carga con la dirección de la instrucción alterna durante la fase de ejecución de instrucciones. Después, cuando llega a la siguiente operación de captura, contiene la dirección de la instrucción en la localidad de la ramificación en vez de la siguiente instrucción secuencial.

El direccionamiento indirecto permite el uso de una dirección indirecta en una instrucción de la computadora. La dirección indirecta, en vez de especificar la ubicación del dato, especifica la dirección en la cual está ubicada la dirección directa, que es la dirección del dato. La fase indirecta es el tiempo durante el cual la unidad de control captura la dirección directa a partir de la dirección indirecta de la memoria. Existe un bit especial en la instrucción que indica si el campo de direcciones contiene una dirección indirecta o una dirección directa.

Durante la fase de ejecución de instrucciones, en el DR se capturan operandos de la memoria (si es necesario), la ALU realiza la operación que especifica el código de operación de la instrucción empleando operandos del acumulador y del DR, y colocando el resultado en el acumulador, y los resultados se almacenan en la memoria (de ser necesario).

Una interrupción⁵ es un procedimiento que permite a la computadora responder a un requisito de procesamiento de alta prioridad distinto del programa en proceso y asíncrono con respecto a éste. La interrupción es iniciada por una señal lógica en una línea específica que se vuelve activa; esto se llama solicitud de interrupción. En respuesta a esta solicitud, la computadora suspende la ejecución del programa y transfiere el control de la CPU a otro programa en la memoria principal llamado manejador de interrupciones. Esta transferencia de control se efectúa durante la fase de interrupción del ciclo de instrucción. Cuando el manejador de interrupciones termina el procesamiento de la interrupción, la computadora prosigue con la siguiente instrucción en el programa regular.

59.3 INSTRUCCIONES

El conjunto de instrucciones⁶ se divide en grupos con base en el modo de direccionamiento que se utilice. La similitud de las instrucciones de un mismo grupo hace posible utilizar una descripción común para todo el grupo. Los siguientes tipos de instrucciones representan los más difundidos en las computadoras actuales.

Instrucciones con indicación de registro. En ellas el o los operandos están todos ubicados en registros de la CPU. Si hay registros generales o registros múltiples de un tipo determinado, la instrucción contendrá campos que identifiquen el o los registros particulares que se utilizarán, así como la operación por efectuar según lo especifique el código de operación. Las instrucciones de esta categoría realizan operaciones aritméticas y lógicas, así como operaciones de ramificación incon-

Tabla 59-1. Conjunto de instrucciones de la PDP-11

Operación	Descripción	Operación	Descripción
ADC	Sumar el acarreo	CLC	Eliminar el código de condición del acarreo
ADD	Sumar	CLN	Eliminar el código de condición negativa
ASL	Desplazamiento aritmético a a izquierda	CLR	Eliminar
ASR	Desplazamiento aritmético a la derecha	CLV	Eliminar el código de condición de desbordamiento
BCC	Ramificar en la eliminación del acarreo	CLZ	Eliminar el código de condición cero
BCS	Ramificar en la colocación del acarreo	CMP	Comparar
BEQ	Ramificar en el signo de igual	COM	Complementar
BGE	Ramificar en el signo de mayor que o igual a	DEC	Disminuir
BGT	Ramificar el signo de mayor que	HALT	Suspender
BHI	Ramificar en el signo de superior	INC	Aumentar
BHIS	En el signo de superior o igual	JMP	Saltar
BIC	Eliminar el bit	JSR	Saltar a la subrutina
BICB	Byte para eliminar el bit	MOV	Mover
BIS	Colocar el bit	MOVB	Mover el byte
BISB	Byte de colocación del bit	NEG	Negar
BIT	Probar el bit	ROL	Girar a la izquierda
BLE	Ramificar en el signo de menor que o igual a	ROR	Girar a la derecha
BLO	Ramificar en el signo de inferior	RTI	Regreso de la interrupción
BLOS	Ramificar en el signo de inferior o igual	RTS	Regreso de la subrutina
BLT	Ramificar en el signo de menor que	SEC	Colocar en el código de condición del acarreo
BMI	Ramificar en el signo menos	SEN	Colocar el código de condición negativa
BNE	Ramificar en el signo de no igual	SEV	Colocar el código de condición de desbordamiento
BPL	Ramificar en el signo más	SEZ	Colocar el código de condición cero
BR	Ramificar	SUB	Restar
BVC	Ramificar en la eliminación del desbordamiento	TST	Probar
BVS	Ramificar en la colocación del desbordamiento	TSTB	Probar el byte

dicional y condicional con datos contenidos en registros, y manejan operaciones relacionadas con las interrupciones.

Instrucciones de referencia a la memoria. En ellas uno o ambos operandos están en la memoria. Si existe un solo operando en la memoria, por lo general hay un segundo operando en un registro. Este tipo de instrucción suele denominarse instrucción de longitud de palabra fija, porque la longitud de ambos operandos es igual al tamaño del registro, es decir, el tamaño del dato procesado por una sola instrucción. La longitud de palabra de la computadora es a menudo igual al tamaño de palabra de la memoria. Los dos operandos de este tipo de instrucción se combinan conforme al código de operación, y el resultado por lo general reemplaza al operando del registro. Este tipo de operación se utiliza para realizar operaciones aritméticas y lógicas con datos que están en la memoria.

Una instrucción que contiene ambos operandos en la memoria suele ser una instrucción de longitud de pala-

bra variable, debido a que el tamaño de los operandos queda especificado por otros campos de la instrucción, no por la arquitectura de la computadora. El resultado de aplicar el código de operación a los operandos por lo general es la sustitución de uno de los operandos contenidos en la memoria. Instrucciones de este tipo suelen emplearse para cambiar datos de un sitio de la memoria a otro, así como para efectuar operaciones aritméticas de longitud variable; la suma de valores decimales codificados en binario es un ejemplo.

Instrucciones de entrada y salida. Provocan la transferencia de datos entre un dispositivo de entrada o salida (I/O) y un registro de la CPU (por lo general el acumulador) o la memoria. Entre las unidades comunes de datos transferidos en una operación individual de I/O se cuentan el bit; el byte, que tiene ocho bits; y la palabra, que generalmente tiene cuatro bytes.

Las instrucciones para una minicomputadora PDP-11⁷ (tabla 59-1), son las típicas del conjunto de instrucciones de una computadora moderna de tamaño mediano.

59.4 ORGANIZACIÓN DE DUCTOS

Los ductos de una computadora⁸ son las trayectorias eléctricas por las cuales se transfieren datos entre registros y otras unidades de la computadora. Un **ducto** es un conjunto de líneas en paralelo por las cuales se pueden transferir simultáneamente varios bits que representen un dato, por ejemplo una palabra. Aquí se describen dos tipos de estructuras de ductos: la cableada y la central.

Estructura de ductos cableada. En ella se hacen conexiones entre registros para permitir la transferencia de datos entre ellos según se requiera para ejecutar las operaciones necesarias. En la figura 59-3 se presenta una estructura de ductos cableada de una computadora común. En esta figura solamente se presentan los ductos y líneas de datos; se omiten las líneas de control.

Estructura de ductos central. Suele emplearse en computadoras relativamente complejadas; tiene una arquitectura más flexible que la estructura cableada. Un ejemplo es el caso de una computadora en la que se emplean registros múltiples de uso general. En la figura 59-4 se muestra este tipo de estructura. Los ductos A SELECT y B SELECT transportan señales de control desde la unidad de control hasta los multicanalizadores que determinan los dos operandos para la ALU. El ducto DESTINATION SELECT (selección

del destino) y la línea ENABLE (de habilitación) del decodificador determinan en qué registro se cargará el resultado (en caso de que ello ocurra). En esta figura no se presentan otras líneas de control que requiere la ALU. Está claro a partir de la figura que es posible combinar el contenido de dos registros cualesquiera mediante cualquiera de las operaciones que la ALU puede realizar, y que puede cargarse el resultado en cualquiera de los registros. La CPU cableada, por otra parte, sólo permite la realización de ciertas transferencias de datos.

59.5 OPERACIÓN DE LA CPU

La operación de la CPU cableada que se ilustra en la figura 59-5 es típica de las computadoras modernas. Las líneas individuales que entran en registros, compuertas o la ALU, cuyas fuentes no aparecen en la figura, representan líneas de control desde la unidad de control. Las funciones de estas líneas se describen a continuación.

La primera fase del ciclo de operación es la de búsqueda (captura) de instrucciones. Esta fase consta de cuatro pulsos de reloj consecutivos durante los cuales ocurren las siguientes acciones:

- **Pulso de reloj 1. SEL (selección)** se coloca en el estado bajo, de manera que la salida del multi-

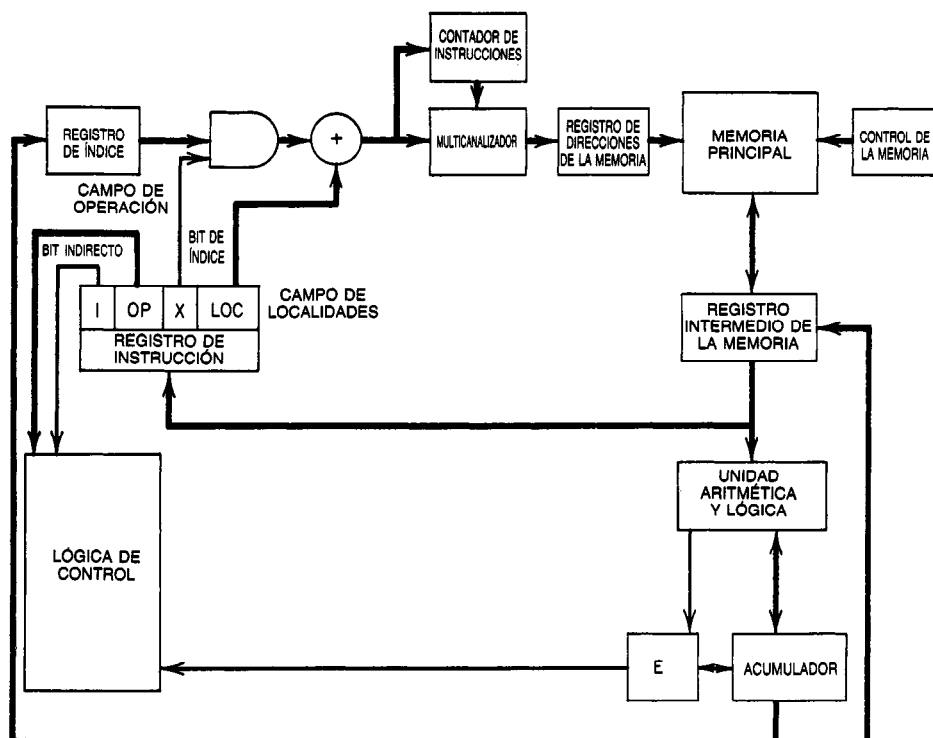


Fig. 59-3. Estructura de ductos cableados. I = bit indirecto; OP = campo de operación; X = bit de índice; LOC = campo de localidades; E = bit de extensión.

