



Los usuarios podrán en cualquier momento, obtener una reproducción para uso personal, ya sea cargando a su computadora o de manera impresa, este material bibliográfico proporcionado por UDG Virtual, siempre y cuando sea para fines educativos y de Investigación. No se permite la reproducción y distribución para la comercialización directa e indirecta del mismo.

Este material se considera un producto intelectual a favor de su autor; por tanto, la titularidad de sus derechos se encuentra protegida por la Ley Federal de Derechos de Autor. La violación a dichos derechos constituye un delito que será responsabilidad del usuario.

Referencia bibliográfica

Levine Gutiérrez, Guillermo. (1989). Elementos de programación. En *Introducción a la computación y a la programación estructurada*. (Pp. 149-209). México: McGraw-Hill.

Introducción a la computación y a la programación estructurada

Segunda edición

Guillermo Levine Gutiérrez

Grupo Micrológica
Universidad Autónoma Metropolitana, Iztapalapa
México.

McGRAW-HILL

MÉXICO • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA • MADRID • NUEVA YORK
PANAMÁ • SAN JUAN • SANTAFÉ DE BOGOTÁ • SANTIAGO • SÃO PAULO
AUCKLAND • HAMBURGO • LONDRES • MILÁN • MONTREAL • NUEVA DELHI • PARÍS
SAN FRANCISCO • SINGAPUR • ST. LOUIS • SIDNEY • TOKIO • TORONTO

Portada: *La máquina de la decidibilidad*, por Ariel Guzik.
(La "decidibilidad" es idea fundamental de la computabilidad,
tema al que se dedica el capítulo 5 de la presente obra.)



UNIVERSIDAD DE GUADALAJARA
UNIDAD DE BIBLIOTECAS
CUCEI

EIC-047923

No. ADQUISICION _____
CLASIFICACION _____
FACTURA DONACION-06 _____
FECHA 29/04/06 _____
EJ. _____ V. _____

INTRODUCCIÓN A LA COMPUTACIÓN Y A LA PROGRAMACIÓN ESTRUCTURADA

Prohibida la reproducción total o parcial de esta obra,
por cualquier medio, sin autorización escrita del editor.

DERECHOS RESERVADOS © 1989, respecto a la segunda edición por
McGRAW-HILL/INTERAMERICANA DE MÉXICO, S.A. DE C.V.
Atlacomulco No. 499-501, Fracc. Ind. San Andrés Atoto
53500 Naucalpan de Juárez, Edo. de México
Miembro de la Cámara Nacional de la Industria Editorial, Reg. Núm. 1890

ISBN 968-422-511-3
(ISBN 968-451-608-8 primera edición)

9012345678 P.E.-89 9087651234

Impreso en México Printed in Mexico

Esta obra se terminó de
imprimir en Octubre de 1994 en
Litográfica Ingramex
Centeno Núm. 162-1
Col. Granjas Esmeralda
Delegación Iztapalapa
09810 México, D.F.

Se tiraron 3500 ejemplares

Contenido

Prólogo a la segunda edición.....	xiii
Prólogo a la primera edición.....	xvii

PRIMERA PARTE

1. Resumen histórico de la computación.....	1
1.1 Antecedentes y razón de ser.....	1
1.2 Generaciones de computadoras.....	8
1.3 Microcomputadoras y computadoras personales.....	13
1.4 Anexo: tecnología de microcomputadoras.....	19
Palabras y conceptos clave.....	25
Ejercicios.....	25
Referencias para el capítulo 1.....	26
2. ¿Cómo funciona una computadora?.....	29
2.1 Introducción.....	29
2.2 El modelo de von Neumann.....	30
Palabras y conceptos clave.....	40
Ejercicios.....	41
Referencias para el capítulo 2.....	42
3. Descripción funcional de un sistema de cómputo.....	43
3.1 El procesador central.....	43
3.2 La memoria central.....	47
3.3 Unidades de entrada y salida.....	49
3.4 Unidades de memoria auxiliar.....	53
3.5 Teleproceso.....	59
3.6 El sistema de cómputo integrado.....	64
3.7 Anexo: estándar internacional para redes.....	68
Palabras y conceptos clave.....	69

Ejercicios	70
Referencias para el capítulo 3	72
4. La programación de sistemas	75
4.1 Lenguaje de máquina	75
4.2 Ensambladores	79
4.3 Macroprocesadores	83
4.4 Cargadores	85
4.5 Compiladores	89
4.6 Sistemas operativos	100
4.7 Utilerías: editores, bases de datos, hojas de cálculo	115
4.8 Inteligencia artificial	122
4.9 Resumen del capítulo	125
4.10 Anexo: cómo se diseña una computadora	130
Palabras y conceptos clave	137
Ejercicios	138
Referencias para el capítulo 4	139

SEGUNDA PARTE

5. Computabilidad	149
5.1 Introducción	149
5.2 El concepto de algoritmo: la máquina de Turing	151
5.3 Lenguajes formales y autómatas	159
5.4 Anexo: visión histórica de la lógica matemática	166
Palabras y conceptos clave	177
Ejercicios	177
Referencias para el capítulo 5	179
6. Elementos de programación	183
6.1 Introducción	183
6.2 Fases de creación de un programa	184
6.3 Herramientas para construir programas	190
6.4 Anexo: lenguajes de programación	204
Palabras y conceptos clave	209
Ejercicios	209
Referencias para el capítulo 6	210
7. Programación estructurada	213
7.1 Introducción	213
7.2 Creación de programas en pseudocódigo	215
7.3 Estructuras adicionales de control	225

7.4	Módulos y subrutinas.....	230
7.5	Técnicas de diseño descendente.....	234
7.6	Documentación y prueba de programas.....	241
7.7	Anexo: elementos de lógica proposicional.....	245
7.8	Anexo: ejemplo de programas codificados en diversos lenguajes.....	248
	Palabras y conceptos clave:.....	268
	Ejercicios.....	268
	Referencias para el capítulo 7.....	270
8.	La codificación en la programación estructurada: FORTRAN y Pascal	275
8.1	Introducción.....	275
8.2	Estructuras fundamentales de control.....	276
8.3	Estructuras adicionales de control.....	287
8.4	Módulos y subrutinas.....	294
8.5	Ejemplo de un diseño completo codificado.....	308
8.6	Manejo de archivos.....	314
	Palabras y conceptos clave.....	342
	Ejercicios.....	343
	Referencias para el capítulo 8.....	344
Apéndice A.	El sistema operativo Unix.....	347
	Referencias sobre Unix.....	370
Apéndice B.	El lenguaje C.....	375
	Referencias sobre el lenguaje C.....	387
Glosario	mínimo.....	391
Nota final		409
Resumen de la bibliografía		411
Índice	temático.....	421

Elementos de programación

6.1 Introducción

Una vez entendidos los principios elementales de funcionamiento de un sistema de cómputo y estudiado el concepto fundamental de algoritmo, pasaremos a una sección más operativa dentro de este curso: aprender a programar una computadora. Se intenta que el lector adquiera algunas habilidades prácticas que le permitirán, con la atención suficiente, escribir programas para computadora correctos, legibles y claros.

Cabe hacer una advertencia: se aprenderá a *programar* una computadora, no a *codificar* un programa. La diferencia entre ambos conceptos es fundamental, y no está entendida del todo en el medio profesional de la computación ni por completo, por desgracia, en el medio académico.

Por *programar* se entiende un proceso mental complejo, dividido en varias etapas. La finalidad de la programación, así entendida, es comprender con claridad el problema que va a resolverse o simularse por medio de la computadora, y entender también con detalle cuál será el procedimiento mediante el cual la máquina llegará a la solución deseada.

La *codificación* constituye una etapa necesariamente posterior a la programación, y consiste en describir, en el lenguaje de programación adecuado, la solución ya encontrada, o sugerida, por medio de la programación. Es decir, primero se programa la solución de un problema y después hay que traducirla a la computadora.

La actividad de programar es, más que nada, conceptual, y su finalidad es intentar definir, cada vez con mayor precisión, acercamientos que resuelvan el problema de manera virtual, es decir, que efectúen una especie de “experimentos mentales” sobre el problema por resolver o simular. El resul-

Diferencias entre
programación y
codificación

tado de tales experimentos constituirá una descripción de los pasos necesarios para encontrar la solución.

Esta descripción, como cualquier otra, estará expresada en un lenguaje determinado. La importancia de la programación consiste en que este lenguaje funciona a la vez como vehículo descriptor y como modelo de la representación dada a la solución; las principales características de ese lenguaje son que es neutro y completo. El primer concepto denota su independencia respecto a alguna máquina en particular, y el segundo se refiere al poder del mismo para expresar cualquier idea computacional.

Un lenguaje así recibe el nombre de **pseudocódigo**, y nuestra primera tarea consiste en entenderlo y aprenderlo, para después aplicarlo a tareas sencillas de programación.

6.2 Fases de creación de un programa

Escribir programas para computadora es una actividad que requiere una buena cantidad de esfuerzo mental y de tiempo. Armados ya de una descripción teórico-conceptual de lo que es una computadora y lo que es un algoritmo, emprendemos el camino de volver realidad la construcción de modelos para representar descripciones de fenómenos o procesos del mundo real.

Esto implica una metodología científica, repetible y comprobable. Se hablará ahora del proceso mental asociado a la construcción de programas para una computadora. Se observará que las primeras fases de este proceso no difieren demasiado de las equivalentes para casi cualquier otra rama de las ciencias básicas, en el sentido de que constituyen un conjunto de pasos bien especificados que se acercan paulatinamente a una solución.

Las fases en la construcción de un programa son, en orden, las siguientes (aunque debe quedar claro que no hay límites tajantes entre el final de una y el inicio de otra*):

- 0) *Entender* el problema.
- 1) Hacer el *análisis* del mismo (a veces este paso se denomina *análisis del sistema*).
- 2) *Programar* el modelo de solución propuesto.
- 3) *Codificarlo*.
- 4) Cargarlo a la computadora para su *ejecución y ajuste*.
- 5) Darle *mantenimiento* durante su tiempo de vida.

El paso cero parece banal, pero deja de parecerlo cuando se piensa en la gran cantidad de proyectos de computación que se desarrollan (y a veces se terminan) sin haber comprendido bien para qué eran, o cuál era el problema

* Compárense estas fases con las propuestas en [TREJ82] (págs. 351-353), por ejemplo.

que supuestamente iban a resolver. Y si, además, se toma en cuenta que los sistemas de programación reales, a diferencia de los ejercicios de carácter didáctico o académico, muchas veces son largos y complejos, e implican la participación de varias personas (a veces decenas o cientos) durante largos periodos, se podrá comprender la importancia de entender con claridad el problema antes de abocar recursos a su solución. El mundo está demasiado poblado de proyectos y sistemas (no sólo de computación) que o no resuelven el problema para el que fueron diseñados, o bien lo hacen pero de una manera sólo aproximada y deficiente*.

No existe un criterio único e infalible para entender con claridad un problema, por lo que nos conformaremos con recomendar medida y claridad en el momento de enfrentarse con uno por vez primera. Al final, de lo que se trata es de crear y mantener una idea clara, un “mapa mental” del problema propuesto, y de ser capaz de abarcarlo de un solo vistazo. Esto obliga a hacer caso omiso de detalles y particularidades operativas en una primera instancia.

Análisis de sistemas

La segunda fase —primera para nosotros— es muy importante; consiste en efectuar un análisis completo del problema o sistema existente, con la finalidad de proponer un modelo para su solución. Está claro que este modelo no puede existir sin que se hayan especificado con claridad todos y cada uno de los componentes estructurales del sistema.

Entendemos por sistema un conjunto estructurado de elementos interrelacionados de alguna manera que puede hacerse explícita. Obsérvese que para la definición del sistema no nos preocupa la función que éste desempeña, ni tampoco exigimos a sus componentes que cooperen entre sí para conseguirlo, ni ninguna otra consideración de carácter animista o teleológico.

Concepto de sistema

Se insiste en los aspectos estructurales porque son la clave para entender y analizar un problema no trivial.

La *estructura* de un sistema es la forma en que están relacionados entre sí sus diversos componentes, de modo que es perfectamente posible tener dos sistemas diferentes con componentes iguales. La diferencia estará en la forma de hacer corresponder unos con otros.

Éste no es el lugar para discutir acerca de lo que son los sistemas, pero sí cabe decir lo que no son. Por desgracia, existe una concepción, demasiado ligera y vulgarizada, de que cualquier cosa puede suponerse como un siste-

* Claro que en este lamentable estado de cosas influyen muchos otros aspectos, además de no entender con claridad el problema antes de intentar su solución, y casi todos ellos tienen que ver con consideraciones de corte psicológico —y sociológico—. Un estudio muy original de algunas de las causas de que las cosas salgan mal se puede encontrar en el divertido libro *El principio de Peter*, de Laurence Peter y Raymond Hull, Plaza Janés, Barcelona, 1971. El inglés Northcote Parkinson se dedicó durante los años de la Segunda Guerra Mundial a estudiar las causas del mal funcionamiento de las organizaciones, y esto dio como resultado las conocidas “leyes de Parkinson”, acerca de las cuales escribió varios libros posteriormente. En los ambientes de ingeniería son muy conocidas también las “leyes de Murphy”, que intentan explicar en una o dos frases la razón de que los proyectos no funcionen como se esperaba.

ma, y ha surgido en consecuencia una fiebre de considerar todo desde “el punto de vista de los sistemas”. Los peligros de tal enfoque son graves, pues pueden llevar a generalizaciones carentes de rigor metodológico y científico. Hay que tener cuidado de llamar sistema solamente a aquellos complejos de elementos en los que su interrelación pueda ser explicitada por medio de un modelo matemático o, por lo menos, a partir de una descripción libre de ambigüedades.

Volviendo a nuestro punto, se debe efectuar el análisis del sistema (o problema) que es candidato a ser “computarizado”. Para esto se dispone de varios enfoques cualitativos, cuya finalidad consiste en proponer el lugar y la función de los componentes aislables del sistema, en términos tanto de los demás como de la función, ahora sí, que será desempeñada por el conjunto.

Un primer enfoque consiste en aplicar una especie de “rejilla mental” y superponerla al sistema, de manera que sus componentes queden englobados en algún elemento de ella. Esta no es otra cosa que una manera estándar de atacar problemas y darles soluciones preestablecidas por la práctica. Este procedimiento se emplea muy a menudo en problemas rutinarios, y propone soluciones que funcionan adecuadamente para la mayoría de los casos tipificables. En ingeniería civil es común aplicar métodos preestablecidos en manuales para resolver problemas de estructuras sencillas. Siempre que un método de esta clase cumpla sus objetivos no deberá haber impedimento para usarlo, pero son realmente pocos los casos complejos donde puede resultar de utilidad.

Existe al menos otro método para problemas dotados de una estructura menos normalizable, y consiste en formular un modelo que se adapte especialmente a la “forma” del problema. Así debe ser, porque se parte de que el problema en estudio no es tipificable; esto significa, en forma figurada, que no tiene una forma preestablecida pues, si la tuviera, habría una entrada en el manual correspondiente que la describiera, junto con una solución estandarizada.

La función del analista de sistemas consiste precisamente en describir el modelo que mejor se adapte a la estructura del problema que se estudia. Un enfoque funcional puede ser adecuado en muchos casos (es decir, hacer el análisis partiendo de la función que cada componente desempeña en el sistema como un todo). Sin embargo, en otro tipo de problemas puede emplearse un análisis dirigido por los datos que maneja el sistema, o por algún otro aspecto que pueda servir de guía.

Función del análisis

El análisis de sistemas en computación es una actividad compleja y altamente dependiente de consideraciones humanas; por tanto, no ha sido aún comprendida en su totalidad dentro de un esquema matemático. Esto quiere decir que las experiencias previas en el análisis son factor primordial en el desarrollo de uno nuevo, y que no existe —a nuestro entender— una manera “segura” de lograr un análisis correcto o productivo en primera instancia, sino que el proceso está sujeto a mejoras, que pueden ser producto de esquemas inductivos o de simples ensayos de prueba y error.

El resultado final de un análisis puede consistir en diagramas que muestren el flujo de la información (no confundirlos con los diagramas de flujo).

Esto es equivalente a un mapa que muestra los diferentes caminos que la información toma dentro del conjunto, junto con una jerarquización de las diversas funciones que el sistema desempeña. El resultado del análisis también puede ser una descripción del funcionamiento del sistema actual (en caso que exista), o de cómo se propone que funcione el nuevo propuesto.

Más adelante se dan ejemplos de pequeños análisis, que desembocarán en programas completamente terminados.

Programación

Una vez hecho el análisis de un sistema se procede a convertirlo en un programa de computadora, que estará escrito en pseudocódigo.

Tal vez el siguiente ejemplo nos acerque a la idea que se desea: pensemos en los procesos que pasan por la mente de un arquitecto cuando conoce el terreno, posiblemente lleno de desniveles y rocas, donde habrá de construir un edificio. ¿no tiene acaso que imaginarse la obra completa y hacer un considerable esfuerzo de abstracción, sin el cual será poco provechoso el trabajo posterior? Éste es el papel que cumple el análisis ya descrito, y que ahora deberá verse plasmado y cristalizado en la forma de un programa.

Un programa está formado, estructuralmente, por dos tipos de componentes: estructuras de control y estructuras de datos. Las siguientes secciones estudian sus características, su ciclo de vida, y su funcionamiento y desarrollo.

Las estructuras de control son las formas que existen para dirigir el flujo de acciones que el procesador efectuará sobre los datos que se manejan en un programa, mismos que están organizados de maneras diversas que son, precisamente, las estructuras de datos.

Las estructuras de control básicas, que se estudian más adelante, son la secuenciación, la selección y la iteración condicional, mientras que las estructuras de datos más comunes son los arreglos (o vectores), listas, cadenas, pilas y árboles, que también se eniplean —aunque no se estudian con profundidad— a lo largo de los siguientes capítulos.

Como componentes no estructurales de un programa se puede mencionar, en orden creciente de complejidad, los enunciados, instrucciones, funciones, subrutinas (o procedimientos) y módulos. De todos ellos se hablará en el transcurso de este capítulo y en el siguiente.

Llamar a estructurales a los anteriores elementos significa que su aparición dentro de un programa obedece a razones guiadas por los componentes que sí lo son: las estructuras de datos y de control. O lo que es igual, lo primero que hay que definir al construir un programa son precisamente sus elementos estructurales, aquellos sin los cuales el programa no es tal.

Y éste es el momento de repetir lo que se dijo en la introducción del curso: se aprenderá primero a programar, no a codificar; nos preocupará un problema estructural y no simplemente instancias particulares por resolver, por lo que no será sorpresa encontrarse ahora con una descripción teórico-

conceptual acerca de la programación. No comenzaremos a preocuparnos de los detalles, siempre enfadosos, de los lenguajes de programación específicos.

De lo que se trata ahora es de definir lo que es un programa, y se propone lo siguiente: un programa es un conjunto de declaraciones de estructuras de datos, seguidas de un conjunto de proposiciones (usando esta palabra en un sentido amplio, que abarca todos los componentes de las estructuras de control). Además, este programa o cadena de símbolos válidos cumple otra condición: está bien formado. Una cadena bien formada (o bien construida) es aquella que está hecha siguiendo las reglas sintácticas (en el sentido definido cuando se habló de los compiladores) de la gramática que produce el lenguaje de computación que se emplea.

Como en este caso no se está hablando de ningún lenguaje de programación en particular, entonces nos referiremos a un programa bien formado cuando sea el producto de la aplicación de ciertas reglas de construcción primitivas. Más adelante se muestra que por medio de estas reglas es posible construir cualquier programa que, por tanto, estará bien formado.

Codificación

Una vez terminada la fase de programación se habrá producido una descripción del modelo propuesto, escrita en pseudocódigo. La razón de ser de ese paso fue disponer de un programa que pueda ser probado mentalmente para averiguar si es correcto en principio, y para determinar en qué grado considera todo el análisis hecho anteriormente. El proceso mediante el cual se llega a un programa esencialmente correcto recibe el nombre de refinamientos progresivos y será estudiado con detalle más adelante.

Sin embargo, un programa en pseudocódigo no es ejecutable en una computadora, por lo que se requiere refinarlo más. El objetivo de estos refinamientos consiste en acercar lo más posible el programa escrito en pseudocódigo a un programa escrito en algún lenguaje de programación particular (como los descritos en el anexo de este capítulo). Esta fase, necesariamente posterior a la de programación, se trata ampliamente en el capítulo 8, con los lenguajes FORTRAN y Pascal. El artículo [WIRN84] muestra en forma concisa la relación que debe existir entre un algoritmo y las estructuras de datos que requiere.

Ejecución y ajuste

Cuando al fin se tiene el programa codificado y compilado llega el momento de ejecutarlo y probarlo “sobre la marcha”. Es decir, permitir que la computadora lo ejecute para evaluar los resultados.

La nociva práctica usual —que tiende a desaparecer— es dedicar poco tiempo a las etapas de análisis y programación y enfocar la atención y los recursos a la codificación, razón por la cual la ejecución de uno de tales progra-

mas estará casi siempre plagada de errores. Existen dos tipos de fallas que es posible encontrar en un programa ya codificado: errores de sintaxis y errores de lógica de programación. Los primeros son relativamente triviales, mientras que los segundos son los causantes de los frecuentes retrasos que sufren los proyectos de programación en todos los niveles de complejidad.

Tipos de errores

En efecto, un error de lógica apunta claramente a omisiones y errores en el modelado que se está tratando de hacer de la realidad. Esto casi siempre se debe a un deficiente análisis o a una programación en pseudocódigo incompleta y apresurada. El grueso del trabajo creativo debe dedicarse precisamente a las etapas que planean y hacen posible la codificación.

La concepción moderna de la prueba de un programa se ha desplazado de la etapa de ejecución a la etapa de programación en pseudocódigo, con las consiguientes ventajas en ahorro de recursos de cómputo utilizados y de tiempo dedicado al cansado ciclo (que a veces parece sin fin) de codificación-compilación-ejecución-corrección-codificación.

Esto no implica, por supuesto, que la metodología propuesta sea infalible o produzca resultados limpios en la primera prueba; significa, sí, que el camino que lleva de la concepción de un sistema hasta su ejecución por medio de una computadora sea más corto y con menos sobresaltos.

Mantenimiento

Si se ha tomado el trabajo de planear cuidadosamente un sistema y de transformarlo en un conjunto bien estructurado de programas y módulos, seguramente tendrá una vida útil prolongada y no se utilizará sólo una o dos veces. Este simple hecho obliga a considerar un esquema de mantenimiento que asegure que el modelo ya sistematizado evolucione a un ritmo parecido al que lo haga la realidad que está siendo simulada. Tal vez llegue el momento en que ese proceso o aspecto de la realidad para el que se construyó el sistema haya cambiado cualitativamente, en cuyo caso se habla del término de la vida útil del sistema.

Vida útil de un sistem

Mientras tanto, sin embargo, hay que ser capaces de hacer alteraciones no estructurales al sistema con costo mínimo en recursos de análisis y programación, lo cual de alguna manera está asegurado si el sistema se ha construido de manera modular y estructurada, y si se dispone de la documentación adecuada que lo describa tanto en su diseño como en su uso. Suele decirse que si un sistema sólo es comprensible por su creador, es un mal sistema.

Esa falta de flexibilidad, además, resulta imposible de tolerar para el caso de sistemas realmente grandes, que son creados incluso por cientos de ingenieros en sistemas y programadores. Se dice, por ejemplo, que el sistema operativo de la serie 360 de IBM requirió cerca de 5000 años-hombre para su desarrollo. No puede ser que un sistema de tal magnitud no tenga previstos cambios y adaptaciones constantes.

Aunque nuestros programas no sean siquiera medianamente grandes, tenemos el compromiso de hacerlos claros y flexibles para que admitan mejoras o sugerencias posteriores.

Ha llegado ya el tiempo de poner manos a la obra y aprender los fundamentos de la programación.

6.3 Herramientas para construir programas

En esta nueva sección se mostrarán algunas de las herramientas existentes para escribir programas, una vez que el problema se ha comprendido y que se ha efectuado un análisis, aunque sea elemental, del mismo.

La primera tarea consiste simplemente en escribir en español una descripción de los pasos necesarios para resolver el problema, con base en el análisis previamente efectuado. A continuación vendrá la fase de traducir esto usando ciertas herramientas computacionales que ahora se describen.

Secuenciación

Lo primero que resalta en una descripción de esta clase es su carácter secuencial. Se inicia con el primer paso, se continúa con el segundo, y así hasta llegar al fin. Por trivial que pueda parecer esta idea siempre debe tenerse como estandarte en la programación. Es decir, que la programación es necesariamente una actividad ordenada y disciplinada, que exige en todo momento una gran cohesión en las actividades mentales tendientes a describir adecuadamente el problema que se desea modelar.

Pero no todos los procesos pueden ser descritos sólo con la primera herramienta mencionada, la secuenciación. Se hace necesaria una que permita tomar decisiones sencillas. Esta nueva construcción primitiva se llama selección, y consiste en evaluar, durante la ejecución, una condición booleana, para decidir cuál de dos caminos escoger a continuación. Una condición se llama *booleana** cuando puede adquirir únicamente dos valores de verdad: falso o verdadero.

Cualquier pregunta que admita tan sólo dos posibles respuestas (sí o no) es booleana. Por ejemplo, “¿Cuántos años tienes?” no es una pregunta booleana, puesto que admite cerca de cien posibles respuestas. Pero la pregunta “¿Tienes 25 años?” sí lo es, puesto que solamente puede ser respondida con un sí o un no.

Selección

Otra característica necesaria para que una pregunta (o condición) sea booleana es que sus dos posibles respuestas sean mutuamente excluyentes. Esto es, por el principio del tercero excluido, o tengo 25 años o no los tengo, pero no existe una posibilidad intermedia.

La construcción primitiva de selección, entonces, dirá algo así: “Evalúa cierta condición booleana. Si el resultado es verdadero, entonces ejecuta la proposición 1; en otro caso, ejecuta la proposición 2”.

Una tercera y última construcción primitiva completará un conjunto que ha demostrado ser completo, en el sentido de que con él es posible escribir

* En honor a George Boole, matemático inglés que propuso todo un sistema de lógica formal que se analizó en el anexo 5.4.

cualquier programa. Esta nueva estructura de control recibe el nombre de iteración condicional. Consiste, como su nombre lo indica, en el planteamiento de una repetición de acciones, gobernada también por una condición booleana.

Iteración condicional

Una iteración condicional dice: “Evalúa cierta condición booleana. Si el resultado es verdadero, ejecuta una proposición y continúa de esta manera mientras la condición siga siendo verdadera”. Está claro que el ciclo se romperá cuando la condición deje de ser verdadera y se vuelva falsa. Es posible, incluso, que el ciclo nunca se efectúe, si desde el principio la condición es falsa.

La proposición más simple sobre la que pueden trabajar las tres estructuras fundamentales de control es el enunciado. Un enunciado será la unidad mínima que se pueda ejecutar. Por ejemplo, se quiere lograr que la variable ALFA adquiera el valor cinco (véase pág. 80), esto se puede representar con el siguiente enunciado (que es en realidad una instrucción de asignación): ALFA = 5.

Otro enunciado puede ser, por ejemplo, “escribe el valor actual de la variable ZETA”: escribe ZETA, que en realidad es una instrucción de entrada/salida.

Representaremos un enunciado cualquiera por medio de la letra *e*, y cuando haya necesidad de diferenciar entre varios, entonces se numerarán: *e*₁, *e*₂, ..., *e*_n.

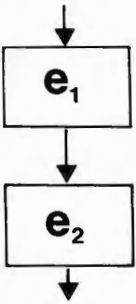
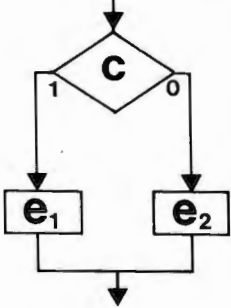
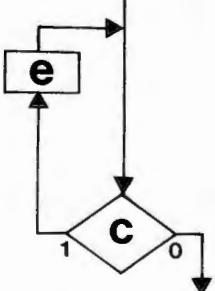
Por otro lado, una condición booleana se representará con la letra mayúscula *C*. Cuando haya que diferenciar entre varias, también serán numeradas. A veces en lugar de un sí se empleará *V* (de verdadero) o el dígito 1. En lugar del no podrá usarse *F* (de falso) o el dígito 0.

Existen por lo menos dos maneras de representar las estructuras de control ya descritas, junto con los enunciados y las condiciones booleanas. Una de ellas es gráfica, mediante esquemas llamados *diagramas de flujo*. La otra es por medios simbólicos, usando pseudocódigo.

Esta última forma, que será la que emplearemos a lo largo de este capítulo y el siguiente, requiere de ciertos símbolos privilegiados, que ya tienen significado preciso y establecido de antemano. A tales indicadores del pseudocódigo se les conoce como palabras clave. Es necesario que exista una palabra clave para la selección y otra para la iteración condicional, y para instrucciones adicionales y otras estructuras de control que veremos después. Por ejemplo, la palabra escribe que se usó líneas atrás es una palabra clave que ya tiene significado preestablecido, a diferencia de la palabra ALFA, que es una variable libre.

En virtud de que las palabras clave son palabras que hablan acerca de otras, adquieren la categoría de metapalabras, razón por la cual se deben distinguir de las que no lo son, para lo que se subrayan.

A continuación se describen las tres estructuras de control, empleando las dos formas ya explicadas:

<p>SECUENCIACIÓN</p>	<p>e_1 e_2 o bien $e_1; e_2$</p>	
<p>SELECCIÓN</p>	<p>si C entonces e_1 otro e_2</p>	
<p>ITERACIÓN CONDICIONAL</p>	<p><u>mientras</u> $(C) e$</p>	

Se pueden observar varias cosas importantes. Existen palabras reservadas para la selección y la iteración condicional, mas no para la secuenciación, porque en tal caso basta simplemente con escribir los enunciados: uno en cada renglón, o en el mismo, pero separados por un punto y coma.

En el pseudocódigo de la selección decidimos aprovechar los espacios en blanco del papel para dibujar la estructura, poniendo en renglones independientes, pero en la misma columna, las dos partes que son mutuamente excluyentes. Esta es una ventaja de la representación en pseudocódigo, ya que permite determinar de inmediato la estructuración del programa fuente.

Una observación primordial ~~es que cada una de estas estructuras de control tiene un solo punto de entrada y un solo punto de salida~~. Esto servirá para armar programas completos uniendo salidas con entradas, para formar cadenas de proposiciones estructuradas en secuencia.

Una sola entrada-una sola salida

Por ejemplo, en el pseudocódigo de la secuenciación, la entrada es la proposición anterior al renglón que comienza con el enunciado e_1 (que no se ve), y la salida es la proposición que está después (que tampoco aparece). Igual sucede con las otras dos.

En los diagramas de flujo la entrada estará arriba del dibujo y la salida estará abajo (aunque, como se verá más adelante, los conceptos de arriba y abajo en los diagramas de flujo son bastante engañosos).

Como ya se dispone de estructuras básicas de control, se procederá a hacer uso de ellas, combinándolas de diversas maneras para escribir programas completos. Para este fin existe una *regla fundamental de composición*, que dice lo siguiente:

Es posible combinar las estructuras de control de secuenciación, selección e iteración condicional, utilizando para tal fin la secuenciación, la selección y la iteración condicional.

Es decir, es posible hacer, por ejemplo, la secuenciación de una secuenciación con una secuenciación, o la selección entre una secuenciación y una iteración condicional.

Si esta regla de composición parece rara es porque es recursiva. Obsérvese que define nuevas estructuras de control (de cualquiera de los tres tipos conocidos) a partir, precisamente, de los tres tipos de estructuras de control.

Como se decía seis palabras atrás, (una definición es recursiva si emplea el *definiens* dentro del *definiendum*) o, en otras palabras, si dice cómo obtener conceptos nuevos empleando para tal fin el mismo concepto que se intenta definir! Este razonamiento parece fallido; parece que no llevará a ninguna parte por ser circular. En realidad, los razonamientos recursivos se encuentran en la base misma de las matemáticas, porque son necesarios para describir conceptos centrales, como el de número.

Un razonamiento recursivo tiene dos partes: la base y la regla recursiva de construcción. La base no es recursiva y es el punto tanto de partida como de terminación de la definición.

El conjunto de los números naturales puede definirse recursivamente así:

Base: El 1 es un número natural. Todo número natural tiene un sucesor.

Regla: El sucesor de un número natural es un número natural*.

Es decir, la base es la existencia del número natural 1. Como ese número existe, debe tener un sucesor (que se llama 2). Por tanto, 2 existe, y como ya es un número natural, entonces debe tener un sucesor, y así, *ad infinitum*.

Esto fue expresado, aunque no exactamente de esta forma, por el gran matemático y lógico italiano Giuseppe Peano (1858-1932).

Regla de formación
de programas

Podría reconsiderarse la regla de formación de estructuras válidas de control de la siguiente manera:

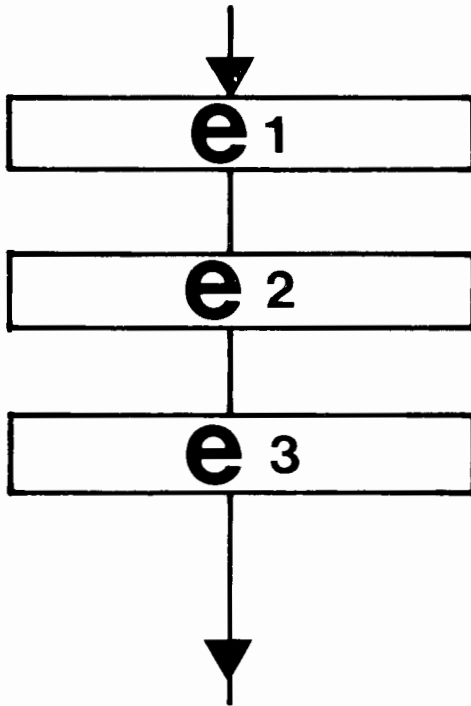
- Base: La secuenciación, la selección y la iteración condicional son estructuras válidas de control que pueden ser consideradas como enunciados.
- Regla: Las estructuras de control que se puedan formar combinando de manera válida la secuenciación, selección e iteración condicional también serán válidas.

Esto da origen al concepto de **programación estructurada**. Un programa estará bien construido si está formado por estructuras de control válidas, de acuerdo con la regla precedente. Las referencias [DAHO72], [MCGC75] y [LINR79] tratan, al igual que parte de nuestro próximo capítulo, de la teoría de la programación estructurada*.

Ya sabemos que la secuenciación de dos enunciados (e_1 y e_2) se forma así: $e_1; e_2$. ¿Cómo se formará la de tres? Pues con una secuenciación de la secuenciación anterior (considerada como un único enunciado compuesto) con el nuevo enunciado e_3 , para entonces obtener: $e_1; e_2; e_3$.

Obsérvese que esta nueva formación sigue teniendo una sola entrada y una sola salida, puesto que es también una secuenciación.

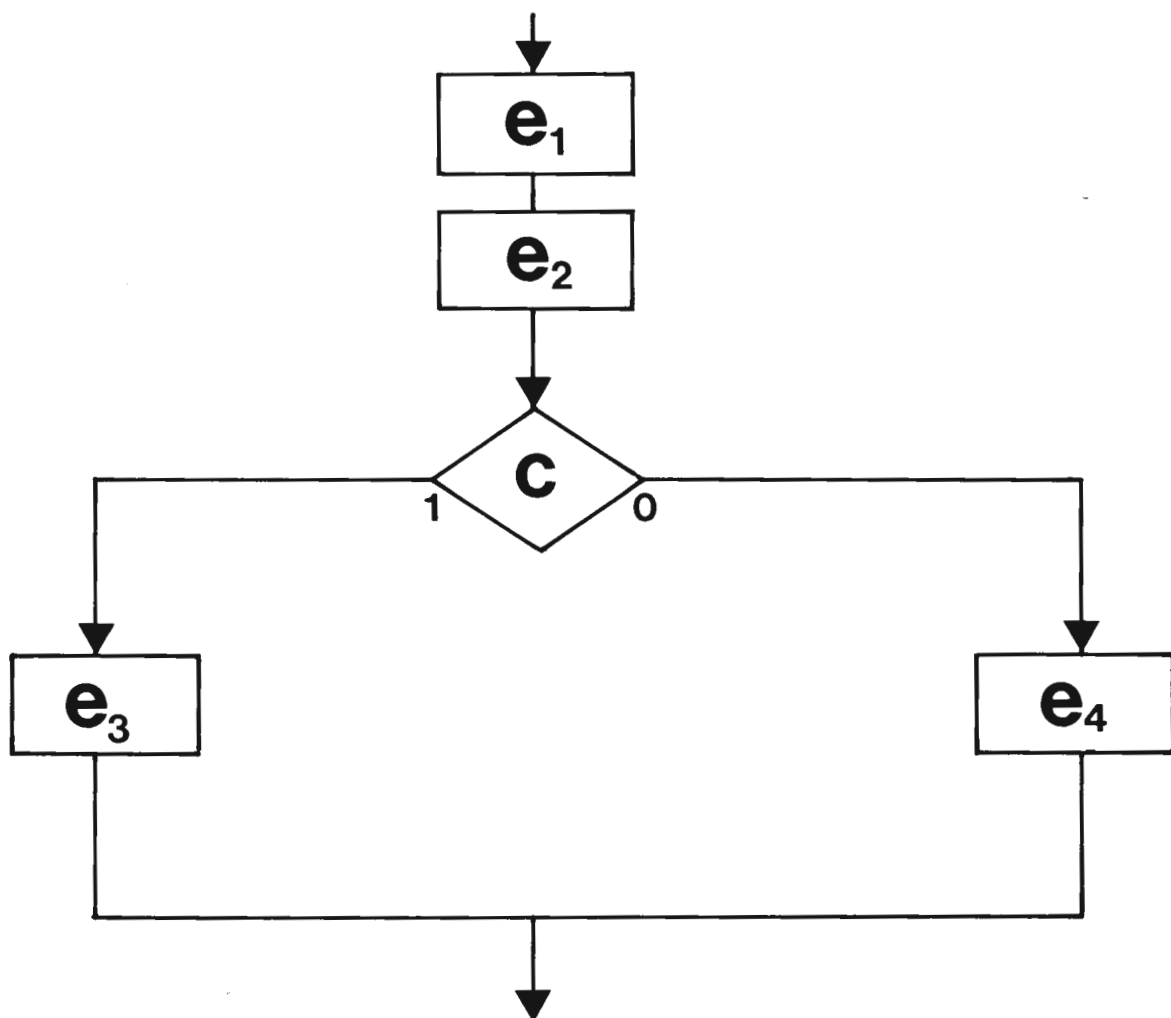
* Los antecedentes teóricos de la programación estructurada se remontan a los inicios de la teoría de la computabilidad, y uno de los primeros resultados, que proponía métodos de normalización para diagramas de flujo, se expuso en un artículo comúnmente considerado el pionero en este campo, "Flow Diagrams, Turing Machines and Languages With Only Two Formation Rules", de Conrado Böhm y Giuseppe Jacopini, publicado en *Communications of the Association for Computing Machinery*, vol. 9, núm. 5, mayo de 1966. El artículo, sin embargo, es completamente teórico, y no tiene una conexión directa con las tareas de programación, pero aun así es señalado como el más importante, y como el que dio origen a lo que se conoce como el "Teorema de la programación estructurada". En el artículo "On Folk Theorems", de David Harel, aparecido también en *Communications of the ACM*, vol. 23, núm. 7, julio de 1980, se hace una incisiva y mordaz crítica a toda esta concepción, un tanto mítica, de los orígenes de la programación estructurada. Un poco para contrarrestar esta visión mitificada de la programación estructurada (y de sus "axiomas"), Donald Knuth escribió un extenso artículo titulado "Structured programming with go to statements", publicado en *Computing Surveys of the ACM*, vol. 6, núm. 4, 1974, en el que explica, con ejemplos, cómo sí es posible (y hasta deseable en algunos casos) emplear la proposición *go to* con provecho dentro de la programación estructurada.



Obtengamos ahora la secuenciación de una secuenciación con una selección:

e_1
 e_2
si C entonces e_3
otro e_4

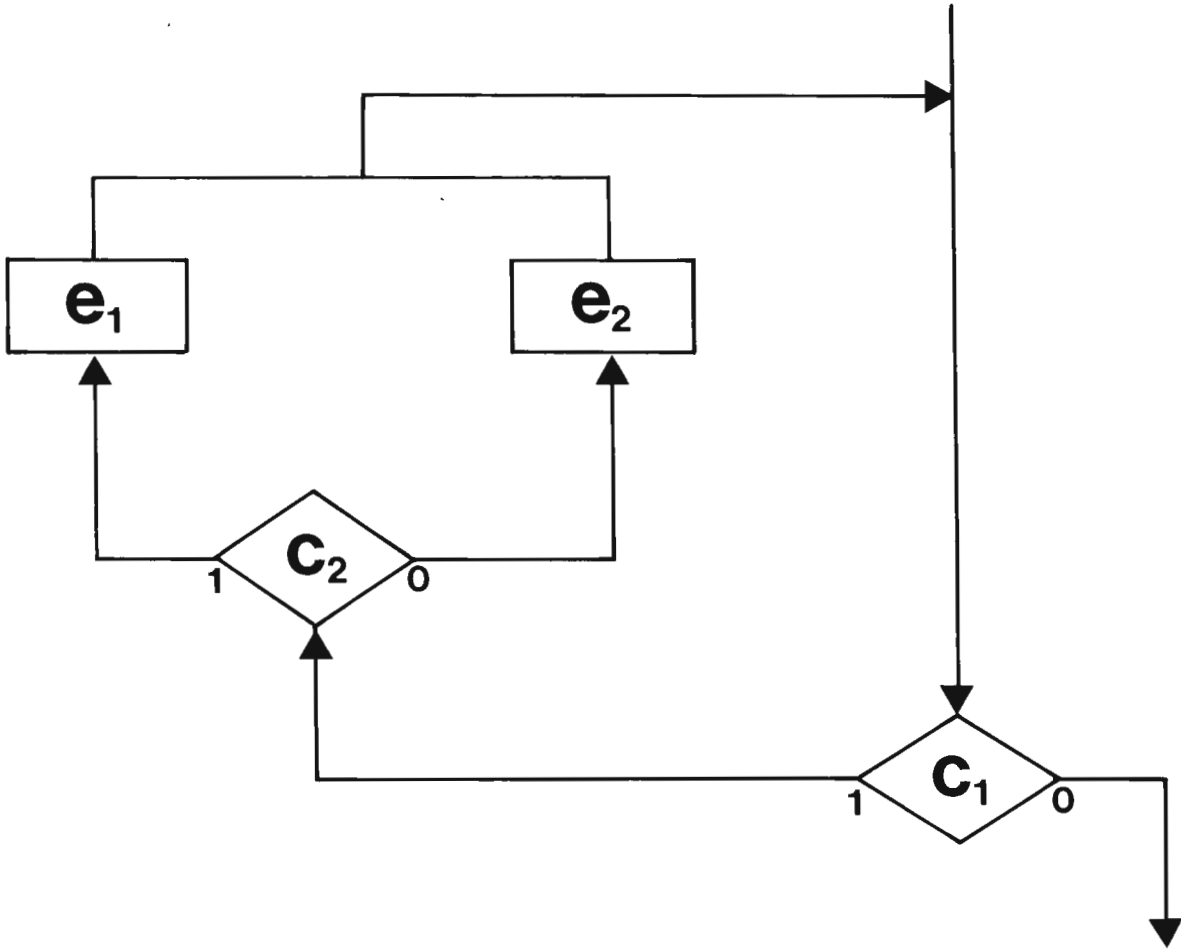
La nueva estructura (secuenciación compuesta) sigue teniendo una sola entrada y una sola salida, y éste es su diagrama de flujo:



Debe estar claro ya que esta otra también es válida, porque se trata de la iteración condicional de una selección:

mientras (C1)
si C2 entonces e1
 otro e2

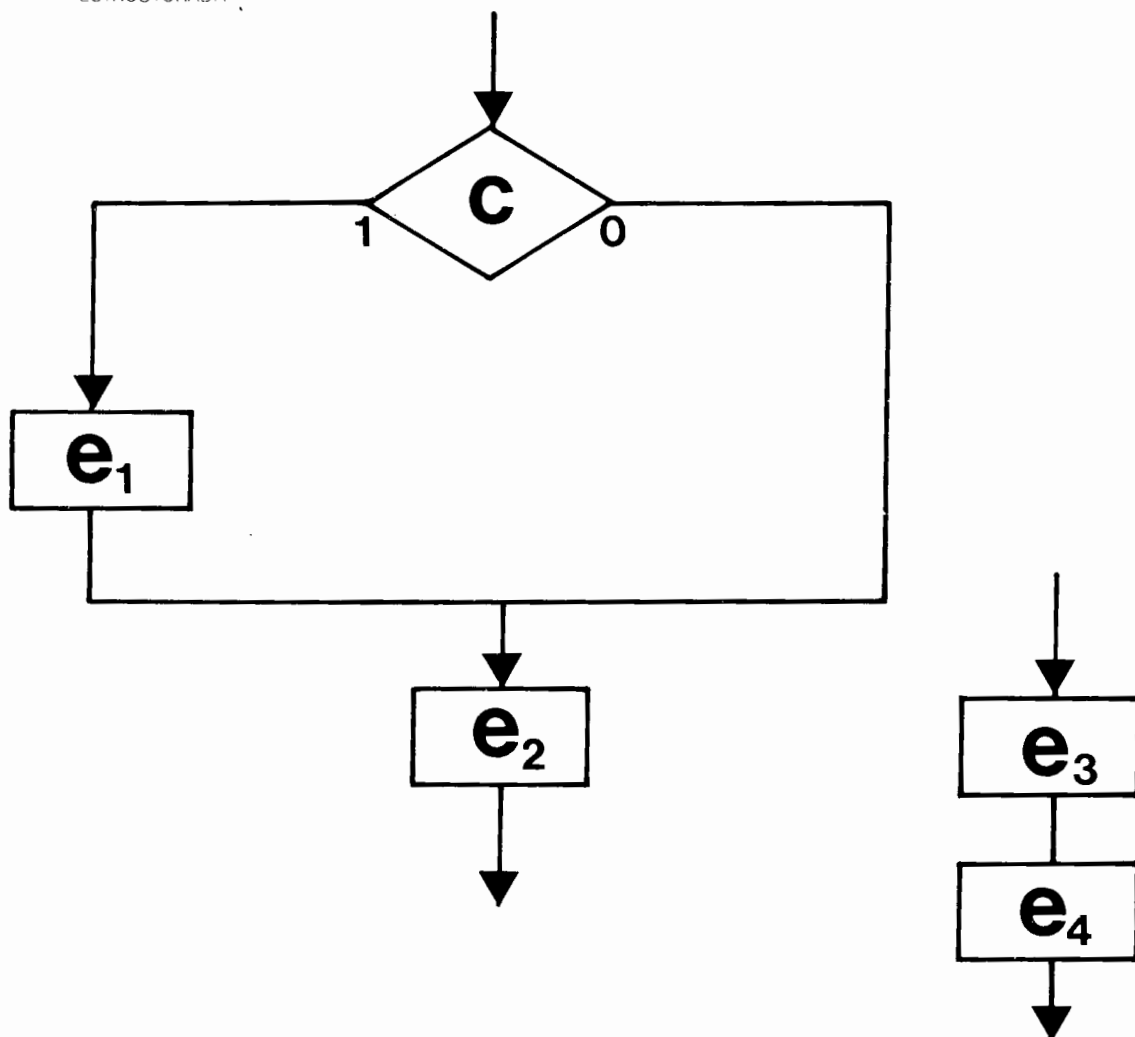
que tiene el siguiente diagrama de flujo:



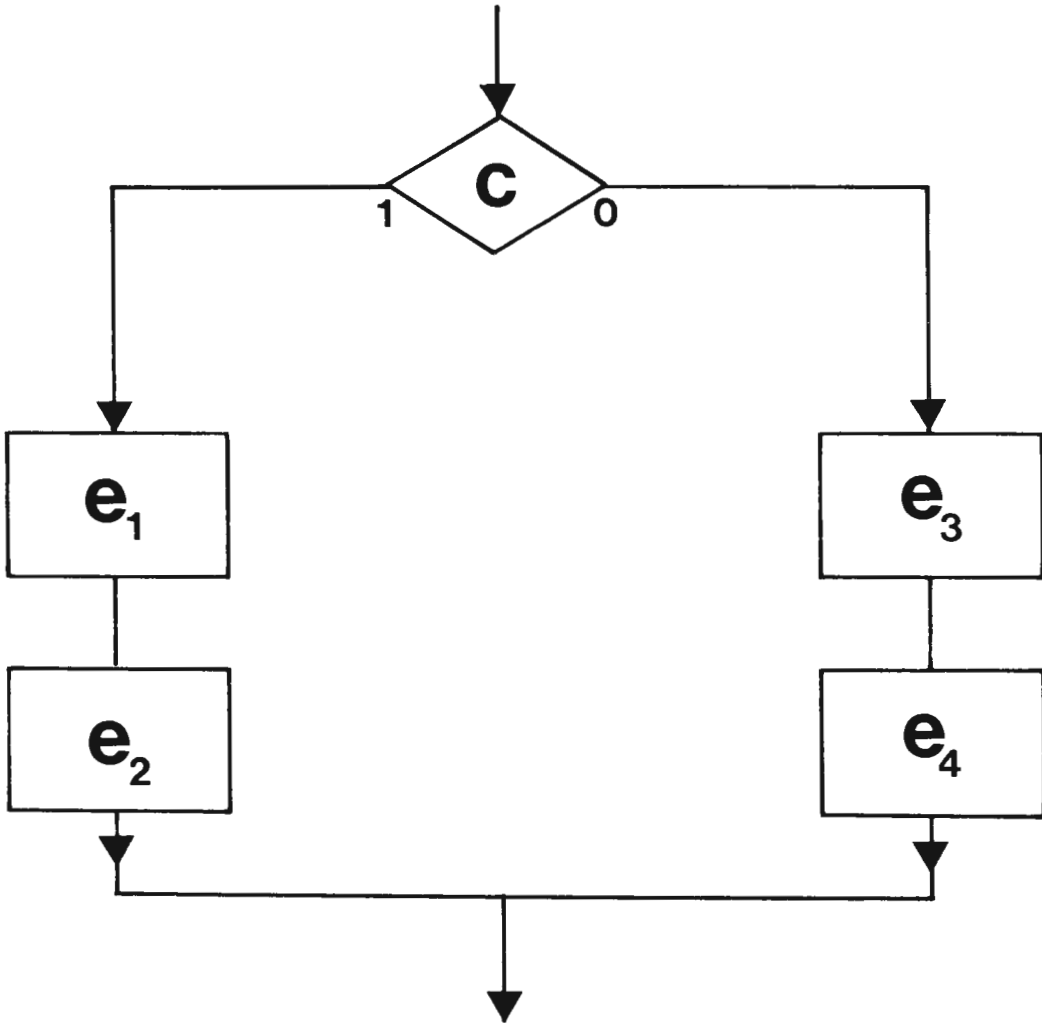
Ahora intentemos hacer la selección de una secuenciación con otra:

si C entonces $e_1; e_2$
otro $e_3; e_4$

Aquí tenemos un problema de ambigüedad: no está claramente definido el alcance del entonces, ni el alcance del otro. O sea, no está claro si e_2 se ejecuta después de e_1 solo si C es verdadera o si, por el contrario, e_2 se ejecuta después de e_1 sin preocupar el valor de C . Si se escoge la segunda posibilidad, el otro está fuera de lugar, porque la estructura quedaría definida así:



en lugar de quedar de esta otra manera, que es la correcta:



Claramente, lo que se requiere es delimitar en forma adecuada el alcance de las proposiciones o estructuras de control.

Diremos que *el alcance sintáctico de una proposición es de una sola proposición*.

¿Qué hacer cuando, como en el ejemplo anterior, se desea que el entonces abarque dos enunciados y no uno solo? Pues se encierran con una especie de paréntesis que los haga aparecer como si fueran un solo enunciado. Estos metaparéntesis son dos nuevas palabras clave del pseudocódigo: comienza y termina. Es importante desde ahora evitar que se confundan estas metapalabras con acciones por ejecutar; en efecto, termina no tiene nada que ver con el hecho de detener la ejecución del procesador o algo por el estilo.

Entonces, la figura anterior deberá leerse en pseudocódigo así:

```

si C entonces comienza
    e1
    e2
    termina
otro comienza
    e3
    e4
    termina

```

Con este expediente queda resuelta la ambigüedad planteada, junto con cualquier otra del mismo tipo que pudiera aparecer.

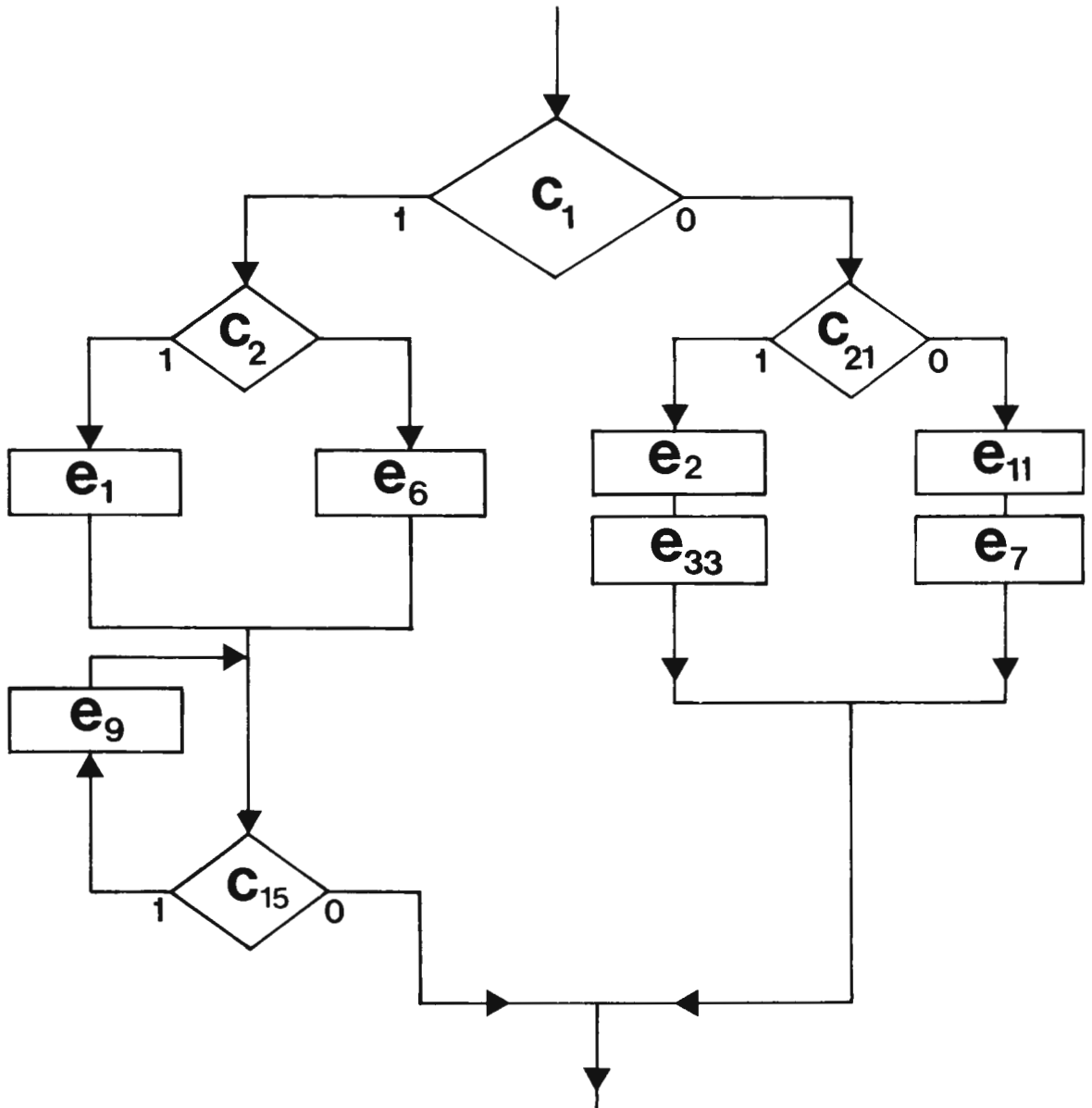
En este otro ejemplo se hace la selección entre la secuenciación de una selección con una iteración condicional y la selección entre una secuenciación y otra. Claro que resulta mucho más fácil leer el pseudocódigo que tratar de entender lo que se acaba de decir.

```

si C1 entonces comienza
    si C2 entonces e1
    otro e6
    mientras (C15)) e9
    termina
otro si C21 entonces comienza
    e2
    e33
    termina
otro comienza
    e11
    e7
    termina

```

El diagrama de flujo correspondiente al pseudocódigo anterior es:



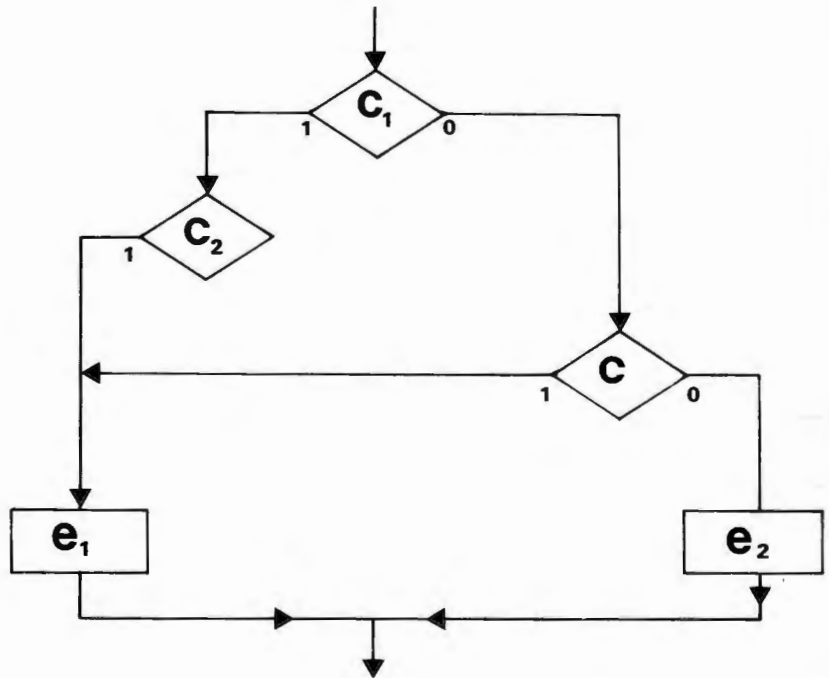
El problema con los diagramas de flujo —que no recomendamos— es que a medida que crece la complejidad (grado de anidamiento) de las proposiciones, también crece el detalle con el que hay que dibujarlos. Esto llega a convertirlos en figuras fraccionadas (pues de otro modo no habría espacio suficiente en la hoja) difíciles de seguir y entender. Además, cuando el diagrama es complejo y tiene proposiciones de tipo mientras, que a veces obligan a que

la entrada y la salida del enunciado asociado estén dibujadas “de cabeza”, entonces sí que resultan oscuros.

Pruebe el lector hacer ejercicios de construcción y anidamiento de estructuras de control para que se dé cuenta por sí mismo.

Si una estructura de control compleja está bien formada (estructurada), entonces seguirá teniendo una sola entrada y una sola salida. Pero la *conversa* no es cierta. O sea, el que una figura tenga una sola entrada y una sola salida no necesariamente significa que esté bien formada, como lo demuestra el siguiente diagrama de flujo no estructurado:

Si está bien formada,
tiene una sola entrada y
una sola salida



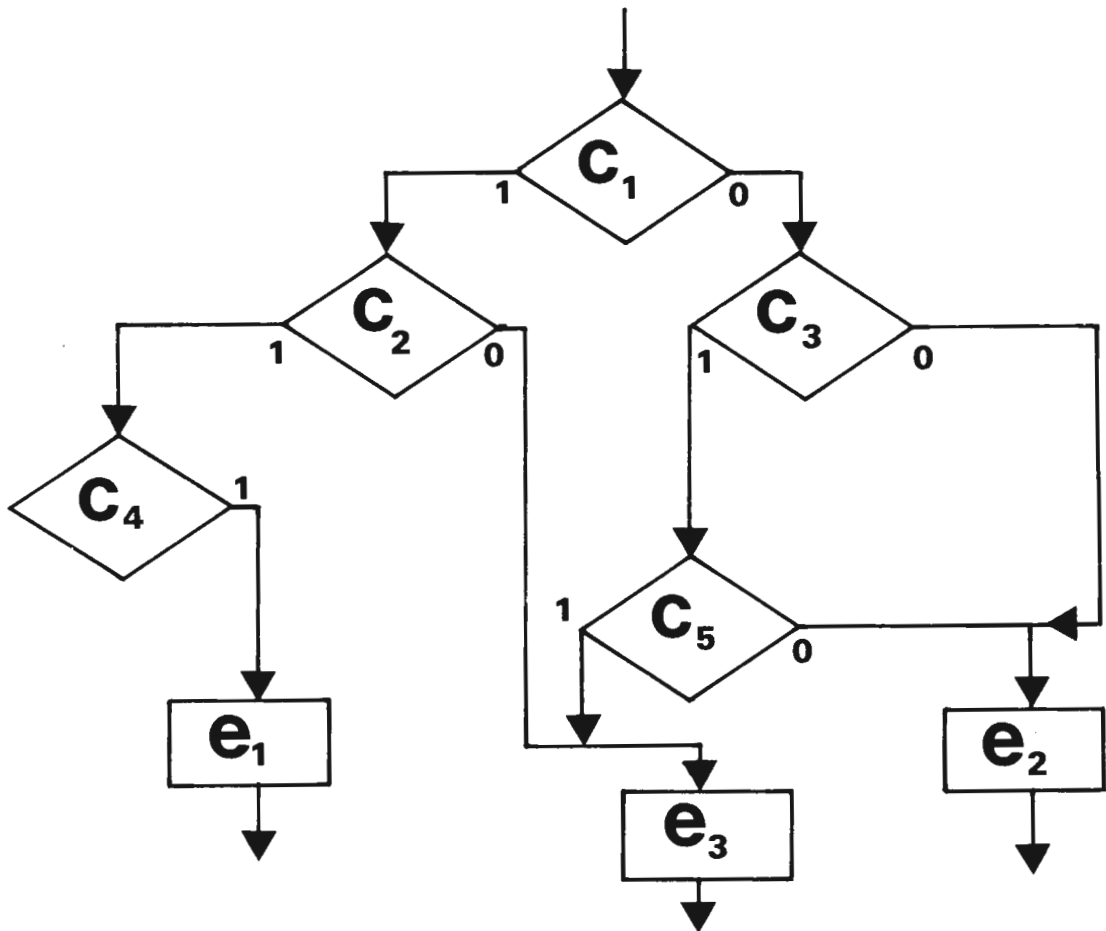
Aprovecharemos lo ya dicho para discutir otra desventaja de los diagramas de flujo que, a nuestro juicio, los vuelve inadecuados para la programación estructurada. Si se observan las figuras de la selección simple y de la iteración condicional simple, es posible creer, erróneamente, que la segunda no es primitiva porque, en apariencia, puede construirse por medio de la primera.

Sin embargo, no es posible construir la iteración condicional a partir de la selección sin tener que recurrir a la flecha que conecta el enunciado de la iteración con su condición, y esta flecha no está definida como válida en las reglas de formación.

El equivalente simbólico de la flecha se llama *go to* (ir a) y está, en términos generales, excluido de la programación estructurada debido a su poder “des-

estructurante'' y caótico sobre los programas (aunque si se usa con cuidado puede llegar a ser de alguna utilidad*).

En efecto, considérese este nuevo diagrama de flujo, donde hay tantas flechas caóticas que la figura resultante ya no tiene estructura definible, y tiene una sola entrada pero varias salidas. Este hecho vuelve imposible interconectar subfiguras análogas, porque entonces ya no habría un criterio único para tal acción, puesto que se rompió la correspondencia una salida-una entrada. El resultado es un programa que no está estructurado, y que no puede ser analizado ni mantenido fácilmente.



* Suele considerarse a la proposición *go to* como no deseable e incluso como peligrosa, y esto no se debe tan sólo a las razones aquí mencionadas, sino a una cierta mística que acompaña a la programación estructurada, y cuyos orígenes pueden encontrarse —entre otras fuentes— en una carta que el conocido autor Esdger Dijkstra envió a la prestigiosa revista *Communications of the ACM* y que apareció publicada en el número de marzo de 1968 con el sugestivo título (puesto por el editor) "La proposición *go to* se considera dañina".

En el siguiente capítulo se emplearán estos nuevos conceptos estructurados para construir una metodología de programación, y se ligarán a las estructuras de datos sencillas, para obtener programas completos, que cumplan su objetivo y sean claros y legibles.

Recomendamos muy especialmente al lector que haga la mayor cantidad posible de ejercicios de programación formal. Es decir, que escriba programas, como los que se han hecho aquí, bien formados, sin preocuparse de para qué puedan servir, o qué problema van a resolver. Este aspecto semántico ("para qué quiero un programa") será analizado más adelante. Por ahora hay que tener la seguridad de que se han entendido cabalmente los conceptos enunciados, y de que se es capaz de manejar con eficacia el anidamiento de estructuras de control, guiados por las reglas recursivas de formación.

Hay que recordar que la base conceptual y matemática de la programación de computadoras indica que los programas pueden ser vistos (si así se deseara) como "teoremas" de un cálculo formal no interpretado, o como cadenas bien formadas de símbolos definidos por medio de reglas libres de ambigüedad. Teniendo esto en cuenta el lector puede dedicarse durante un tiempo a producir programas bien estructurados, haciendo caso omiso por lo pronto de su función semántica.

6.4 Anexo: lenguajes de programación*

En la sección 4.5 se estudiaron los compiladores; aquí se mencionan algunos de los lenguajes de programación más importantes y usuales.

Ada, que debe su nombre a Ada Lovelace, asistente de Babbage en el desarrollo de la máquina analítica, es un intento más por tener un único lenguaje de programación que sea de uso verdaderamente universal. El diseño de este lenguaje, que tomó varios años, fue auspiciado por el departamento de defensa de los Estados Unidos, que exige, desde 1981, que toda la programación que se desarrolle internamente esté escrita en Ada; el grupo que desarrolló Ada estuvo a cargo de Jean Ichbiah. Definitivamente se trata de un lenguaje de características avanzadas, que incluye manejo dinámico de memoria y recursividad, así como facilidades integradas para manejo de programación concurrente, pero la realidad es que no ha tenido la difusión prevista, y es sólo uno más en la gran familia de lenguajes de programación. Más aun, Ada ha causado una gran polémica en términos académicos, porque se arguye que el lenguaje (y su correspondiente compilador) es demasiado grande y poco elegante, y que la tendencia moderna en lenguajes de programación es que sean pequeños y funcionales.

* Consúltense los libros [BARN86], [SAMJ69] y [TUCA87] para una referencia amplia sobre muchos lenguajes de programación, así como el artículo [TESL84]. Véase también el anexo 7.8, que contiene ejemplos de diversos programas escritos en los lenguajes BASIC, C, COBOL, FOREST, FORTRAN y Pascal.

ALGOL (*ALGO*rithmic *L*anguage) Poderoso lenguaje de aplicaciones científicas que surge en la década de 1960. Este lenguaje, diseñado por un comité internacional con sede en Europa, fue el primero con sintaxis definida de manera formal y matemática, y puede ser considerado como el iniciador de la familia de lenguajes de programación estructurada. Existe una versión más moderna, usada en algunas universidades (y de amplio uso en Europa), llamada ALGOL 68.

Un programa en ALGOL consiste en módulos conocidos como *procedures*, que pueden estar anidados y pueden ser llamados recursivamente. Como el manejo de memoria es dinámico, las dimensiones de los arreglos pueden variar durante la ejecución.

BASIC (*B*eginners *A*ll *p*urpose *S*ymbolic *I*nstruction *C*ode) Lenguaje dedicado fundamentalmente a la programación de máquinas pequeñas. Fue diseñado en 1964 por John Kemeny y Thomas Kurtz, del Dartmouth College, en New Hampshire, Estados Unidos. Existe un gran número de versiones diferentes, por lo que se podría decir que es casi el único de los lenguajes de programación no estandarizado.

Un programa en BASIC consta de un conjunto de renglones numerados (aunque hay versiones que ya no requieren esto). Se puede llamar a subrutinas, pero algunos tipos de BASIC no permiten el paso de parámetros. El manejo de memoria es estático, aunque con algunos compiladores, que requieren un subsistema durante la ejecución, es posible que las dimensiones de los arreglos varíen en forma dinámica.

Ha habido gran controversia acerca de si BASIC debe enseñarse como primer lenguaje de programación; un punto de vista sostiene que sí, porque es sencillo y fue originalmente diseñado para ese fin, mientras que el otro argumenta que con las prácticas actuales de programación estructurada, no sólo es un lenguaje pobre, sino que además tiene efectos contraproducentes. Un autor muy conocido llegó a hacer la cáustica observación de que BASIC “causa daño cerebral en los que aprenden a programar con él”.

En el anexo 7.8 hay un ejemplo de un programa codificado en BASIC.

C Lenguaje especializado en la programación de sistemas. Diseñado a principios de la década de 1970 por Dennis Ritchie, de los Laboratorios Bell, en New Jersey, Estados Unidos, se emplea para escribir compiladores y sistemas operativos; actualmente buena parte de ellos se escribe en C.

El poder de este lenguaje estriba en que el código que produce es bastante similar al que un programador lograría si escribiera en ensamblador, con la enorme ventaja de ser un lenguaje de alto nivel. Es posible, por ejemplo, asignar registros de la UCP durante la compilación, y llamar a rutinas y subsistemas del sistema operativo desde el programa, sobre todo cuando se emplea desde el sistema operativo Unix.

Un programa en C consta de módulos que pueden ser llamados de manera recursiva, pero no anidados. Su manejo de memoria es dinámico.

En el anexo 7.8 hay un ejemplo de un programa codificado en C, y en dos apéndices al final del libro se hace una semblanza del lenguaje C y del sistema operativo Unix.

COBOL (*CO*mmon *B*usiness *O*riented *L*anguage) Uno de los primeros lenguajes de programación. Fue diseñado en 1958 por la almirante Grace Hopper dentro de un comité norteamericano llamado CODASYL (*CO*nference on *DA*ta *SY*stems *L*anguages), y se sigue usando mucho para aplicaciones comerciales y administrativas.

Un programa en COBOL consta de cuatro “divisiones” (IDENTIFICATION, ENVIRONMENT, DATA y PROCEDURE), que tienen que ser especificadas siempre. Los programas tienden a ser grandes y están llenos de palabras y frases cortas en inglés, por lo que son bastante legibles, si bien no muy estructurados. Su manejo de memoria es estático, aunque la mayoría de las versiones incluyen subsistemas de manejo dinámico de información en disco magnético.

A veces se suele considerar a COBOL como un mal lenguaje de programación, anticuado y engorroso, cuando en realidad el problema estriba en la deficiente preparación en computación por parte de sus adeptos. Desde hace muchos años se ha augurado la desaparición de este lenguaje, pero parece que esto no sucede, sobre todo en el entorno de las grandes computadoras y centros de cómputo. Existen excelentes libros sobre programación estructurada y diseño modular en este lenguaje, y en términos de productividad sería sumamente provechoso elevar el nivel de conceptualización computacional de un conjunto de analistas y programadores de COBOL. En el anexo 7.8 hay un ejemplo de un programa codificado en COBOL.

FORTRAN (*FOR*mula *TRAN*slation) fue el primer lenguaje de programación —diseñado en 1957 por John Backus, de IBM— y su principal aplicación es en las áreas de ingeniería. Se sigue empleando bastante en casi cualquier centro de cómputo. Existen variantes nuevas, obtenidas por medio de preprocesadores (Ratfor, FOREST), y una nueva edición estandarizada, apta para programación estructurada, llamada FORTRAN 77.

Al igual que con COBOL, ha habido multitud de predicciones de que este lenguaje va a desaparecer muy pronto de los centros de cómputo, pero no ha sido así, aunque ciertamente su uso ha disminuido, sobre todo porque en muchas escuelas ya se enseña Pascal en su lugar.

Un programa en FORTRAN puede estar constituido por un programa principal y varias subrutinas. Su manejo de memoria es estático, por lo que no se pueden tener arreglos o vectores que cambien de tamaño durante la ejecución.

En el capítulo 8 se muestran las principales características de FORTRAN y se aplican a la programación estructurada.

Modula-2 Lenguaje reciente (1979), fue diseñado por el creador de Pascal, Niklaus Wirth, e incluye algunas mejoras sobre su antecesor, así como un concepto un tanto diferente para la creación de sistemas de programación. Recibe su nombre de la importancia que le adjudica al manejo de unidades lógicas de programa en forma de módulos. Dispone de formas novedosas de manejo de paso de parámetros, y de operadores para “importar” y “exportar” datos entre módulos, lo cual le da gran flexibilidad y permite la

compilación separada e independiente de rutinas que colaborarán entre sí al momento de la ejecución.

En el anexo 7.8 hay un ejemplo de un programa codificado en Modula-2.

Pascal (diseñado en 1970 por Niklaus Wirth, del Instituto Tecnológico de Zurich, en Suiza) es un lenguaje de programación que se ha vuelto muy popular, sobre todo para máquinas pequeñas. Existe una versión bastante conocida, de la Universidad de California en San Diego, y otra que debe su éxito a un compilador (para computadoras personales) que funciona a enorme velocidad.

Un programa en Pascal consiste en módulos que pueden ser anidados y llamados recursivamente. El lenguaje permite la definición de estructuras de datos que van más allá de las tradicionales (entero, real, etc.), por lo que se presta bastante para la programación estructurada. Existen varias versiones de un Pascal extendido que incluyen operaciones para control de procesos en paralelo, llamado Pascal concurrente.

Su autor desarrolló posteriormente el lenguaje Modula-2, que incorpora algunas mejoras y corrige algunos de los errores conceptuales de Pascal.

En el capítulo 8 se muestran las principales características de Pascal y se aplican a la programación estructurada.

PL/I (*Programming Language I*) es un lenguaje de programación muy amplio y extenso que IBM propuso en 1965 como alternativa para trabajos científicos y comerciales. Aunque no es de uso generalizado, sigue siendo bastante utilizado en ciertos ambientes de cómputo, y es considerado como un lenguaje de programación bueno y completo.

Los compiladores de PL/I son grandes, por lo que a veces se trabaja con subconjuntos del lenguaje para lograr resultados eficientes.

Un programa en PL/I consiste en un procedimiento principal, dentro del cual se pueden anidar bloques. Su manejo de memoria puede ser estático o totalmente dinámico y recursivo. Las versiones completas permiten la programación de “tarefas” (*tasks*) que simulan multiprocesamiento.

RPG (*Report Program Generator*) es un lenguaje diseñado por IBM para producir, como su nombre lo indica, informes administrativos y comerciales. Éste no es realmente un lenguaje de programación genérico, ni es un vehículo adecuado para hacer buena programación. Los programas en RPG no son legibles ni claros, y tienen que ser codificados en hojas especiales. Existe una versión actualizada llamada RPG-III.

Existen muchos otros lenguajes, algunos de ellos de aplicaciones especiales.

APL (*A Programming Language*) es otro ejemplo de un poderoso lenguaje de aplicación especial y matemática, usado sobre todo en máquinas IBM, y que data de 1961. Supuestamente es de uso general, pero la realidad es que su manejo no es muy conocido. También es un medio de representación de no-

tación matemática que pone más énfasis en la definición de formas que en la de procedimientos. Su autor, Kenneth Iverson, explica: “el lenguaje está basado en una unificación y extensión consistentes de las notaciones matemáticas existentes, y en una extensión sistemática de un pequeño conjunto de operaciones matemáticas y lógicas sobre vectores, matrices y árboles”.

LISP (*LIS*t *P*rocessing), es un lenguaje muy usado en la comunidad académica dedicada a investigación en inteligencia artificial. Maneja en forma dinámica conjuntos llamados listas, que el programador construye por medio de elementos primitivos llamados átomos. Es, en esencia un medio para representar funciones parcialmente recursivas en matemáticas, y con él se pueden definir funciones complejas y evolutivas (porque se desarrollan siguiendo esquemas formales). El lenguaje fue diseñado por John McCarthy a principios de la década de 1960, y durante muchos años estuvo ligado con las labores que se desarrollan en el Instituto Tecnológico de Massachusetts (MIT), hasta que fue reconocido como el lenguaje de la inteligencia artificial, puesto que últimamente le disputa Prolog.

En el anexo 7.8 hay un ejemplo de un programa codificado en LISP.

Logo es un lenguaje basado en los mismos principios de LISP, pero expresados en otra forma, por lo que es mucho más fácil de manejar, aunque menos poderoso. Fue diseñado por un educador y matemático del MIT, Seymour Papert. Ha adquirido cierta popularidad para computadoras personales porque hay versiones que producen gráficas de manera casi natural, y se prestan, por tanto, para aplicaciones de tipo educativo.

Prolog (*PRO*gramming *LOG*ic) forma parte de un enfoque de programación diferente del mostrado en este libro, que considera que esta actividad puede ser vista desde una perspectiva de aplicación de funciones lógicas sobre predicados, y que está basado en el cálculo proposicional. Este lenguaje, creado en 1972 por Alan Colmerauer y Philippe Roussel en la Universidad de Marsella, Francia, ha sido adoptado por el proyecto de la quinta generación de computadoras emprendido por Japón, como vehículo de creación de los sistemas y programas para manejo de inteligencia artificial.

En el anexo 7.8 hay un ejemplo de un programa codificado en Prolog.

FORTH es un lenguaje para microcomputadoras que intenta combinar características del lenguaje ensamblador con una metodología de diseño de software conocida como “código hilvanado”, que resulta de interés para algunas aplicaciones, en las que reemplaza al lenguaje ensamblador.

Otros lenguajes dignos de mención son **SNOBOL**, para manipulaciones simbólicas, diseñado en los Laboratorios Bell; **Simula**, lenguaje para simulación de procesos diseñado en Noruega, y **Smalltalk**, lenguaje orientado a objetos (es decir, que el programador no enfoca su atención en variables o estructuras de control, sino en los objetos sobre los que trabaja el programa), diseñado en el centro de investigaciones de Xerox, en California.

Palabras y conceptos clave

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

PROGRAMACIÓN	ESTRUCTURAS DE DATOS	ITERACIÓN CONDICIONAL
CODIFICACIÓN	ENUNCIADOS	REGLA DE COMPOSICIÓN
PSEUDOCÓDIGO	SELECCIÓN	SECUENCIACIÓN
ANÁLISIS DE SISTEMAS	DIAGRAMA DE FLUJO	PALABRA CLAVE
ESTRUCTURAS DE CONTROL	INSTRUCCIONES	PROGRAMACIÓN ESTRUCTURADA

Ejercicios

1. Escriba cinco programas en pseudocódigo y tradúzcalos a sus diagramas de flujo equivalentes.
2. Haga cinco diagramas de flujo estructurados y tradúzcalos a pseudocódigo.
3. ¿Se puede traducir un diagrama de flujo cualquiera a pseudocódigo? ¿Por qué?
4. Los enunciados (e_1, \dots, e_n) y las condiciones (C_1, \dots, C_n) que se han empleado en los programas en pseudocódigo de este capítulo han sido puramente formales, es decir, carecen de contenido porque no representan nada. Tome cualquiera de los programas ya existentes y sustituya sus enunciados y condiciones por aseveraciones válidas en algún contexto, para obtener un programa con significado y bien formado.

Por ejemplo, con los enunciados “encender el horno”, “esperar un minuto”, “mezclar los ingredientes” (aunque es evidente que éste se debe descomponer en varios más), y las condiciones “¿la temperatura es de 300 grados?” y “¿ya está cocido?” se puede describir en pseudocódigo el proceso de cocinar un pastel, desde encender el horno y esperar a que se caliente, hasta que esté terminado.

5. Describa con detalle en pseudocódigo varios procesos usuales de la vida diaria, por ejemplo, entrar en una tienda y comprar un producto (lo cual implica determinar si lo tienen, si el precio es el correcto, formarse en la cola de la caja, etc.), o llegar a un destino en un automóvil o en transporte colectivo.